Open Group Standard

Service-Oriented Architecture Ontology
Version 2.0

Open Group Standard

Service-Oriented Architecture Ontology, Version 2.0

ISBN: 1-937218-50-8

Document Number: C144

Published by The Open Group, April 2014.

Comments relating to the material contained in this document may be submitted to:

The Open Group, Apex Plaza, Forbury Road, Reading, Berkshire, RG1 1AX, United Kingdom

or by electronic mail to:

ogspecs@opengroup.org

Contents

Preface

The Open Group

The Open Group is a global consortium that enables the achievement of business objectives through IT standards. With more than 400 member organizations, The Open Group has a diverse membership that spans all sectors of the IT community – customers, systems and solutions suppliers, tool vendors, integrators, and consultants, as well as academics and researchers – to:

- Capture, understand, and address current and emerging requirements, and establish policies and share best practices
- Facilitate interoperability, develop consensus, and evolve and integrate specifications and open source technologies
- Offer a comprehensive set of services to enhance the operational efficiency of consortia
- Operate the industry's premier certification service

Further information on The Open Group is available at www.opengroup.org.

The Open Group publishes a wide range of technical documentation, most of which is focused on development of Open Group Standards and Guides, but which also includes white papers, technical studies, certification and testing documentation, and business titles. Full details and a catalog are available at www.opengroup.org/bookstore.

Readers should note that updates – in the form of Corrigenda – may apply to any publication. This information is published at www.opengroup.org/corrigenda.

This Document

This document is The Open Group Standard for Service-Oriented Architecture Ontology. It has been developed and approved by The Open Group.

Trademarks

Acknowledgements

The Open Group gratefully acknowledges all contributors to the SOA Ontology project, and in particular the following individuals:

- Edward Altman, Eli Lilly
- Jim Amsden, IBM
- Horia Balog, Telus
- Stuart Boardman, Getronics
- Brian R. Bokor, IBM
- Rex Brooks, Starbourne
- Abby H. Brown, Intel
- Anthony L. Carrato, IBM
- John Colgrave, IBM
- Eric Dabbaghchi, MITRE
- Mark Delaney, Penn State University
- Michele Deo, HP
- Awel Dico, Bank of Montreal
- Chris Greenslade, CLARS
- Ed Harrington, Architecting-the-Enterprise
- Claus T. Jensen, IBM
- Srikanth Kappagantula, HP
- Radha Kasibhatla, HP
- Heather Kreger, IBM
- Milena Litoiu, CGI
- Ovace Mamnoon, HP
- E.G. Nadhan, HP
- Miroslav Novak, HP
- Mike Pasco, IBM

- Rajiv Ranjan, HP
- Kay Sampaongern, The Boeing Company
- Todd J. Schneider, Raytheon
- Jerome Sonnenberg, Harris Corporation
- Andras Szakal, IBM
- Ahmad R. Yaghoobi, The Boeing Company
- Liang-Jie Zhang, IBM

While many of the above made strong contributions, special thanks are due to Claus T. Jensen, who was responsible for the technical direction and much of the wording of the published standard.

Referenced Documents

The following documents are referenced in this standard:

- Business Process Modeling Notation (BPMN), Version 1.1, Object Management Group; available from www.omg.org.
- Beyond Concepts: Ontology as Reality Representation, Barry Smith; available from http://ontology.buffalo.edu/bfo/BeyondConcepts.pdf.
- Definition of SOA: The Open Group; available from www.opengroup.org/soa/soa/def.htm#_Definition_of_SOA.
- IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description of Software-intensive Systems (adopted by ISO/IEC JTC1/SC7 as ISO/IEC 42010:2007); available from standards.ieee.org.
- IETF RFC 2119: Key Words for use in RFCs to Indicate Requirement Levels, March 1997; refer to www.ietf.org.
- ISO/IEC 42010:2007: Systems and Software Engineering – Recommended Practice for Architectural Description of Software-intensive Systems; available from www.iso.org.
- Navigating the SOA Open Standards Landscape Around Architecture (W096), White Paper published by The Open Group, November 2009.
- OASIS Reference Model for Service-Oriented Architecture, Version 1.0, Organization for the Advancement of Structured Information Standards (OASIS); available from www.oasis-open.org.
- OWL Web Ontology Language Reference, W3C Recommendation, 10 February 2004, World-Wide Web Consortium; available from www.w3.org/TR/owl-ref.
- Service-Oriented Architecture Modeling Language (SoaML), Object Management Group; available from www.omg.org.
- The Open Group Architecture Framework (TOGAF), The Open Group; available from www.opengroup.org.
- What is an Ontology? Stanford University; available from www-ksl.stanford.edu/kst/what-is-an-ontology.html.

1 Introduction

1.1 Objective

The purpose of this standard is to contribute to The Open Group mission of Boundaryless Information Flow, by developing and fostering common understanding of Service-Oriented Architecture (SOA) in order to improve alignment between the business and information technology communities, and facilitate SOA adoption.

It does this in two specific ways.

First, it defines the concepts, terminology, and semantics of SOA in both business and technical terms, in order to:

- Create a foundation for further work in domain-specific areas
- Enable communications between business and technical people
- Enhance the understanding of SOA concepts in the business and technical communities
- Provide a means to state problems and opportunities clearly and unambiguously to promote mutual understanding

Secondly, it potentially contributes to model-driven SOA implementation.

The ontology is designed for use by:

- Business people, to give them a deeper understanding of SOA concepts and how they are used in the enterprise and its environment
- Architects, as metadata for architectural artifacts
- Architecture methodologists, as a component of SOA meta-models
- System and software designers for guidance in terminology and structure

1.2 Overview

This standard defines a formal ontology for Service-Oriented Architecture (SOA). SOA is an architectural style that supports service-orientation. This is the official definition of SOA as defined by The Open Group SOA Work Group. For full details, see www.opengroup.org/soa/soa/def.htm#_Definition_of_SOA.

The ontology is represented in the Web Ontology Language (OWL) defined by the World-Wide Web Consortium (W3C). OWL has three increasingly expressive sub-languages: OWL-Lite,

OWL-DL, and OWL-Full. (See www.w3.org/2004/OWL for a definition of these three dialects of OWL.) This ontology uses OWL-DL, the sub-language that provides the greatest expressiveness possible while retaining computational completeness and decidability.

The ontology contains classes and properties corresponding to the core concepts of SOA. The formal OWL definitions are supplemented by natural language descriptions of the concepts, with graphic illustrations of the relations between them, and with examples of their use. For purposes of exposition, the ontology also includes UML diagrams that graphically illustrate its classes and properties of the ontology. The natural language and OWL definitions contained in this specification constitute the authoritative definition of the ontology; the diagrams are for explanatory purposes only. Some of the natural language terms used to describe the concepts are not formally represented in the ontology; those terms are meant in their natural language sense.

This standard uses examples to illustrate the ontology. One of these, the car-wash example, is used consistently throughout to illustrate the main concepts. (See Chapter 8 for the complete example.) Other examples are used *ad hoc* in individual sections to illustrate particular points.

A graphically compressed visualization of the entire ontology is shown below (in Figure 1).

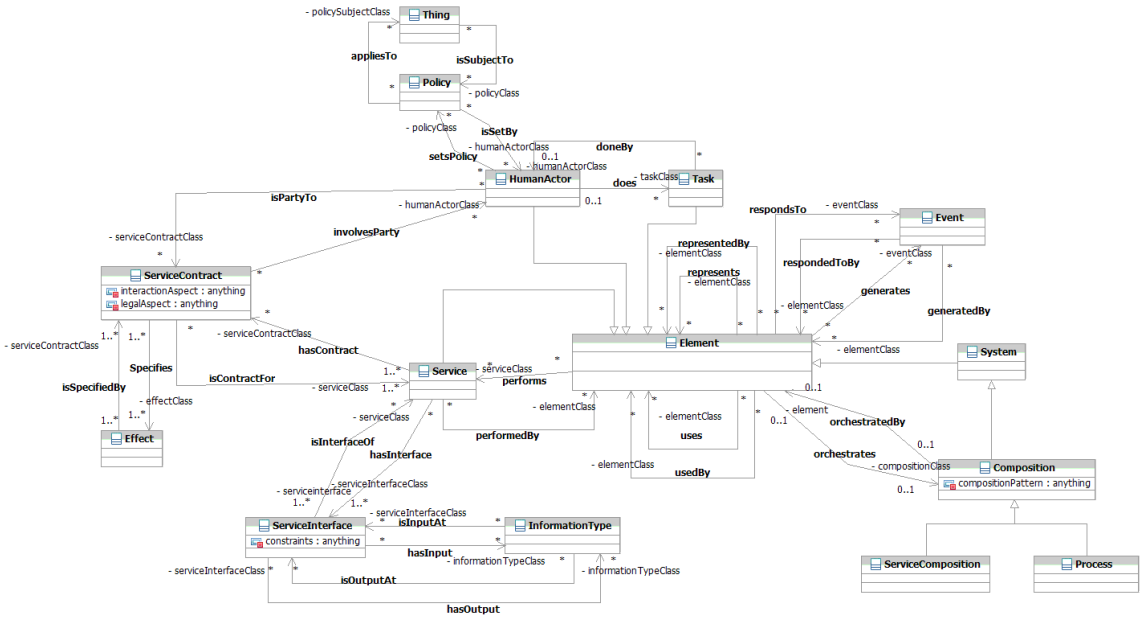

Figure 1: SOA Ontology – Graphical Overview

The concepts illustrated in this figure (Figure 1) are described in the body of this standard.

The class hierarchy is as follows (see Figure 2).

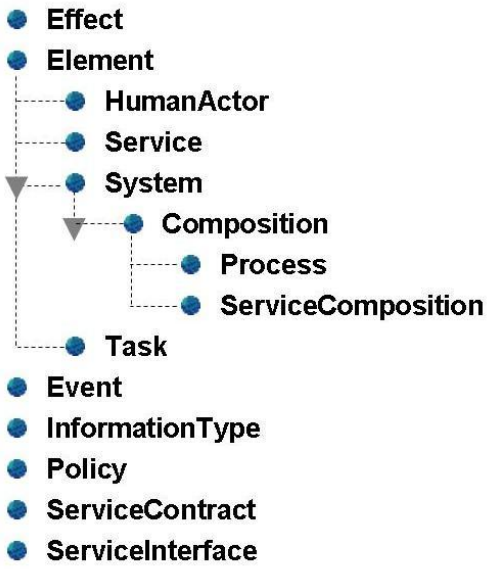

Figure 2: SOA Ontology – Class Hierarchy

The remainder of this standard is structured as follows:

- This chapter provides an introduction to the whole standard.
- Chapters 2 through 7 provide the formal definitions (OWL and natural language) of the terms and concepts included in the ontology.
- Chapter 8 contains the complete car wash example that is used as a common example throughout.
- Chapter 9 contains an additional elaborate example utilizing most of the classes in the ontology.
- Appendix A contains the formal OWL definitions of the ontology, collected together.
- Appendix B describes the relation of this ontology to other work.
- Appendix C contains a relationship matrix that details the class relationships implied by the OWL definitions of the ontology.

1.3 Applications

The SOA ontology specification was developed in order to aid understanding, and potentially be a basis for model-driven implementation.

To aid understanding, this specification can simply be read. To be a basis for model-driven implementation, it should be applied to particular usage domains and application to example usage domains will aid understanding.

The ontology is applied to a particular usage domain by adding SOA OWL class instances of things in that domain. This is sometimes referred to as "populating the ontology". In addition, an application can add definitions of new classes and properties, can import other ontologies, and can import the ontology OWL representation into other ontologies.

The ontology defines the relations between terms, but does not prescribe exactly how they should be applied. (Explanations of what ontologies are and why they are needed can be found in, for example, Beyond Concepts: Ontology as Reality Representation and What is an Ontology?) The examples provided in this standard are describing one way in which the ontology could be applied in practical situations. Different applications of the ontology to the same situations would nevertheless be possible. The precise instantiation of the ontology in particular practical situations is a matter for users of the ontology; as long as the concepts and constraints defined by the ontology are correctly applied, the instantiation is valid.

1.4 Conformance

There are two kinds of applications that can potentially conform to this ontology. One is other OWL-based ontologies (typically extensions of the SOA ontology); the other is a non-OWL application such as a meta-model or a piece of software.

A conforming OWL application (derived OWL-based ontology):

- Must conform to the OWL standard
- Must include (in the OWL sense) the whole of the ontology contained in Appendix A of this standard
- Can add other OWL constructs, including class and property definitions
- Can import other ontologies in addition to the SOA ontology

A conforming non-OWL application:

- Must include a defined and consistent transform to a non-trivial subset of the ontology contained in Appendix A of this standard
- Can add other constructs, including class and property definitions
- Can leverage other ontologies in addition to the SOA ontology

1.5 Terminology

The words and phrases MUST, REQUIRED, SHALL, MUST NOT, SHALL, NOT, SHOULD, RECOMMENDED, SHOULD NOT, NOT RECOMMENDED, and MAY are used in this standard with the meanings defined in IETF RFC 2119.

Furthermore, the meaning of the word opaque (used in the later definition of the concept Element) is defined to indicate that any possible internal structure of something is invisible to an external observer.

1.6 Typographical Conventions

Bold font is used for OWL class, property, and instance names where they appear in section text.

Italic strings are used for emphasis and to identify the first instance of a word requiring definition.

OWL definitions and syntax are shown in `fixed-width font`.

An unlabeled arrow in the illustrative UML diagrams means subclass.

1.7 Future Directions

It is anticipated that this will be a living document that will be updated as the industry evolves and SOA concepts are refined. Future versions of this ontology may include additional core concepts.

Also, this ontology can be used as a core for domain-specific ontologies that apply to the use of SOA in particular sectors of commerce and industry. The Open Group does not currently plan to develop such ontologies, but encourages other organizations to do so to meet their needs.

2 System and Element

2.1 Introduction

System and *element* are two of the core concepts of this ontology. Both are concepts that are often used by practitioners, including the notion that systems have elements and that systems can be hierarchically combined (systems of systems). What differs from domain to domain is the specific nature of systems and elements; for instance, an electrical system has very different kinds of elements than an SOA system.

In the ontology only elements and systems within the SOA domain are considered. Some SOA sub-domains use the term *component* rather than the term element. This is not contradictory, as any component of an SOA system is also an element of that (composite) system.

This chapter describes the following classes of the ontology:

- **Element**
- **System**

In addition, it defines the following properties:

- **uses** and **usedBy**
- **represents** and **representedBy**

2.2 The Element Class

```
<owl:Class rdf:about="#Element">
</owl:Class>
```

An *element* is an opaque entity that is indivisible at a given level of abstraction. The element has a clearly defined boundary. The concept of *element* is captured by the **Element** OWL class, which is illustrated below (in Figure 3).

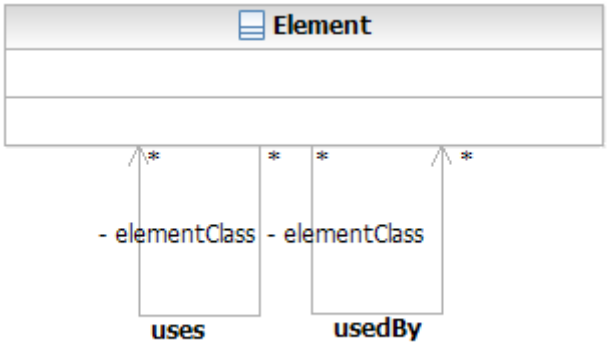

Figure 3: The Element Class

In the context of the SOA ontology we consider in detail only functional elements that belong to the SOA domain. There are other kinds of elements than members of the four named subclasses (**System**, **HumanActor**, **Task**, and **Service**) described later in this ontology. Examples of such other kinds of elements are things like software components or technology components (such as Enterprise Service Bus (ESB) implementations, etc.).

2.3 The uses and usedBy Properties

```
<owl:ObjectProperty rdf:about="#uses">
  <rdfs:domain rdf:resource="#Element"/>
  <rdfs:range rdf:resource="#Element"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#usedBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#uses"/>
  </owl:inverseOf>
</owl:ObjectProperty>
```

Elements may use other elements in various ways. In general, the notion of some element using another element is applied by practitioners for all of models, executables, and physical objects. What differs from domain to domain is the way in which such use is perceived.

An element uses another element if it interacts with it in some fashion. Interacts here is interpreted very broadly ranging through, for example, an element simply being a member of (used by) some system (see later for a formal definition of the **System** class), an element interacting with (using) another element (such as a service; see later for a formal definition of the **Service** class) in an *ad hoc* fashion, or even a strongly coupled dependency in a composition (see Section 5.2 for a formal definition of the **Composition** class). The **uses** property, and its inverse **usedBy**, capture the abstract notion of an element using another. These properties capture not just transient relations. Instantiations of the property can include “uses at this instant”, “has used”, and “may in future use”.

For the purposes of this ontology we have chosen not to attempt to enumerate and formally define the multitude of different possible semantics of a *uses* relationship. We leave the semantic interpretations to a particular sub-domain, application, or even design approach.

2.4 Element – Organizational Example

Using an organizational example, typical instances of **Element** are organizational units and people. Whether to perceive a given part of an organization as an organizational unit or as the set of people within that organizational unit is an important choice of abstraction level:

- Inside the boundary of the organizational unit we want to express the fact that an organizational unit uses the people that are members of it. Note that the same person can in fact be a member of (be used by) multiple organizational units.
- Outside the boundary the internal structure of an organizational unit must remain opaque to an external observer, as the enterprise wants to be able to change the people within the organizational unit without having to change the definition of the organizational unit itself.

This simple example expresses that some elements have an internal structure. In fact, from an internal perspective they are an organized collection of other simpler things (captured by the **System** class defined below).

2.5 The System Class

```
<owl:Class rdf:about="#System">
  <owl:disjointWith>
    owl:Class rdf:about="#Task"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Service"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Element"/>
  </rdfs:subClassOf>
</owl:Class>
```

A *system* is an organized collection of other things. Specifically things in a system collection are instances of **Element**, each such instance being used by the system. The concept of *system* is captured by the **System** OWL class, which is illustrated below (in Figure 4).

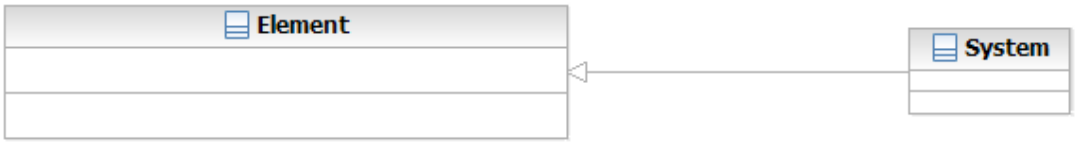

Figure 4: The System Class

This definition of System is heavily influenced by IEEE Std 1471-2000, adopted by ISO/IEC JTC1/SC7 as ISO/IEC 42010:2007: Systems and Software Engineering – Recommended Practice for Architectural Description of Software-intensive Systems.

In the context of the SOA ontology we consider in detail only functional systems that belong to the SOA domain. Note that a fully described instance of **System** should have by its nature (as a collection) a *uses* relationship to at least one instance of **Element**.

Since **System** is a subclass of **Element**, all systems have a boundary and are opaque to an external observer (black box view). This excludes from the **System** class structures that have no defined boundary. From an SOA perspective this is not really a loss since all interesting SOA systems do have the characteristic of being possible to perceive from an outside (consumer) perspective. Furthermore, having **System** as a subclass of **Element** allows us to naturally express the notion of systems of systems – the lower-level systems are simply elements used by the higher-level system.

At the same time as supporting an external viewpoint (black box view, see above) all systems must also support an internal viewpoint (white box view) expressing how they are an organized collection. As an example, for the notion of a service this would typically correspond to a service specification view *versus* a service realization view (similar to the way that SoaML defines services as having both a black box/specification part and a white box/realization part).

It is important to realize that even though systems using elements express an important aspect of the **uses** property, it is not necessary to "invent" a system just to express that some element uses another. In fact, even for systems we may need to be able to express that they can use elements outside their own boundary – though this in many cases will preferably be expressed not at the system level, but rather by an element of the system using that external **Element** instance.

System is defined as disjoint with the **Service** and **Task** classes. Instances of these classes are considered not to be collections of other things. **System** is specifically not defined as disjoint with the **HumanActor** class since an organization in many cases is in fact just a particular kind of system. We choose not to define a special intersection class to represent this fact.

2.6 System – Examples

2.6.1 Organizational Example

Continuing the organizational example from above, we can now express that an organizational unit as an instance of **System** has the people in it as members (and instances of **Element**).

2.6.2 Service Composition Example

Using a service composition example, services **A** and **B** are instances of **Element** and the composition of **A** and **B** is an instance of **System** (that uses **A** and **B**). It is important to realize that the act of composing is different than composition as a thing – it is in the latter sense that we are using the term composition here.

See also below for a formal definition of the concepts of service and service composition (and a repeat of the example in that more precise context).

2.6.3 Car Wash Example

Consider a car wash business. The company as a whole is an organizational unit and can be instantiated in the ontology in the following way:

- **CarWashBusiness** is an instance of **System**.
- **Joe** (the owner) is an instance of **Element** and used by (owner of) **CarWashBusiness**.
- **Mary** (the secretary) is an instance of **Element** and used by (employee of) **CarWashBusiness**.
- **John** (the pre-wash guy) is an instance of **Element** and used by (employee of) **CarWashBusiness**.
- **Jack** (the washing manager and operator) is an instance of **Element** and used by (employee of) **CarWashBusiness**.

2.7 The represents and representedBy Properties

```
<owl:ObjectProperty rdf:about="#represents">
  <rdfs:domain rdf:resource="#Element"/>
  <rdfs:range rdf:resource="#Element"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#representedBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#represents"/>
  </owl:inverseOf>
</owl:ObjectProperty>
```

The environment described by an SOA is intrinsically hierarchically composite (see also Section 5.2 for a definition of the **Composition** class); in other words, the elements of SOA systems can be repeatedly composed to ever higher levels of abstraction. One aspect of this has already been addressed by the **uses** and **usedBy** properties in that we can use these to express the notion of systems of systems. This is still a very concrete relationship though, and does not express the concept of architectural abstraction. We find the need for architectural abstraction in various places such as a role representing the people playing that role, an organizational unit representing the people within it (subtly different from that same organizational unit using the people within it, as the **represents** relationship indicates the organizational unit as a substitute interaction point), an architectural building block representing an underlying construct (for instance, important to enterprise architects wanting to explicitly distinguish between constructs and building blocks), and an Enterprise Service Bus (ESB) representing the services that are accessible through it (for instance, relevant when explicitly modeling operational interaction and dependencies). The concept of such an explicitly changing viewpoint, or level of abstraction, is captured by the **represents** and **representedBy** properties illustrated below (in Figure 5).

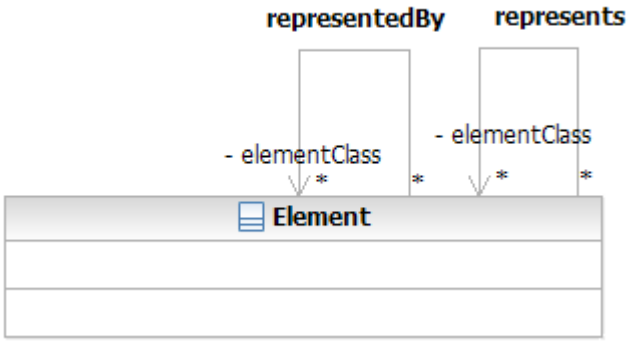

Figure 5: The represents and representedBy Properties

It is important to understand the exact nature of the distinction between using an element (E1) and using another element (E2) that represents E1. If E1 changes, then anyone using E1 directly would experience a change, but someone using E2 would not experience any change.

When applying the architectural abstraction via the **represents** property there are three different architectural choices that can be made:

- An element represents another element in a very literal way, simply by hiding the existence of that element and any changes to it. There will be a one-to-one relationship between the instance of **Element** and the (different) instance of **Element** that it represents. A simple real-world example is the notion of a broker acting as an intermediary between a seller (that does not wish to be known) and a buyer.
- An element represents a particular aspect of another element. There will be a many-to-one relationship between many instances of **Element** (each of which represents a different aspect), and one (different) instance of **Element**. A simple real-world example is the notion that the same person can play (be represented by) many different roles.
- An element is an abstraction that can represent many other elements. There will be a one-to-many relationship between one instance of **Element** (as an abstraction) and many other instances of **Element**. A simple real-world example is the notion of an architectural blueprint representing an abstraction of many different buildings being built according to that blueprint.

Note that in most cases an instance of **Element** will represent only one kind of thing. Specifically, an instance of **Element** will typically represent instances of at most one of the classes **System**, **Service**, **HumanActor**, and **Task** (with the exception of the case where the same thing is both an instance of **System** and an instance of **Actor**). See later sections for the definitions of **Service**, **HumanActor**, and **Task**.

2.8 Examples

2.8.1 Organizational Example

Expanding further on the organizational example, assume that a company wants to form a new organizational unit O1. There are two ways of doing this:

- Define the new organization directly as a collection of people P1, P2, P3, and P4. This means that the new organization is perceived to be a leaf in the organizational hierarchy, and that any exchange of personnel means that its definition needs to change.
- Define the new organization as a higher-level organizational construct, joining together two existing organizations O3 and O4. Coincidentally, O3 and O4 between them may have the same four people P1, P2, P3, and P4, but the new organization really doesn't know, and any member of O3 or O4 can be changed without needing to change the definition of the new organization. Furthermore, any member of O3 is intrinsically *not* working in the same organization as the members of O4 (in fact need not even be aware of them) – contrary to the first option where P1, P2, P3, and P4 are all colleagues in the same new organization.

In this way the abstraction aspect of the **represents** property induces an important difference in the semantics of the collection defining the new organization. Any instantiation of the ontology can and should use the **represents** and **representedBy** properties to crisply define the implied semantics and lines of visibility/change.

2.8.2 Car Wash Example

Joe chooses to organize his business into two organizational units, one for the administration and one for the actual washing of cars. This can be instantiated in the ontology in the following way:

- **CarWashBusiness** is an instance of **System**.
- **AdministrativeSystem** is an instance of **System**.
- **Administration** is an instance of **Element** that represents **AdministrativeSystem** (the opaque organizational unit aspect, *aka* ignoring anything else about **AdministrativeSystem**).
- **CarwashBusiness** uses (has organizational unit) **Administration**.
- **CarWashSystem** is an instance of **System**.
- **CarWash** is an instance of **Element** that represents **CarWashSystem** (the opaque organizational unit aspect, *aka* ignoring anything else about **CarWashSystem**).
- **CarWash** is a member of **CarWashBusiness**.
- **Joe** (the owner) is an instance of **Element** and now used by **AdministrationSystem**.
- **Mary** (the secretary) is an instance of **Element** and now used by **AdministrationSystem**.

- **John** (the pre-wash guy) is an instance of **Element** and now used by **CarWashSystem**.
- **Jack** (the wash manager and operator) is an instance of **Element** and now used by **CarWashSystem**.

3 HumanActor and Task

3.1 Introduction

People, organizations, and the things they do are important aspects of SOA systems. **HumanActor** and **Task** capture this as another set of core concepts of the ontology. Both are concepts that are generic and have relevance outside the domain of SOA. For the purposes of this SOA ontology we have chosen to give them specific scope in that tasks are intrinsically atomic (corresponding to, for instance, the Business Process Modeling Notation (BPMN) 2.0 definition of *task*) and human actors are restricted to people and organizations.

This chapter describes the following classes of the ontology:

- **HumanActor**
- **Task**

In addition, it defines the following properties:

- **does** and **doneBy**

3.2 The HumanActor Class

```
<owl:Class rdf:about="#HumanActor">
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Element"/>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:about="#Task"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Service"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#ServiceContract"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#ServiceInterface"/>
  </owl:disjointWith>
</owl:Class>
```

A *human actor* is a person or an organization. The concept of *human actor* is captured by the **HumanActor** OWL class, which is illustrated below (in Figure 6).

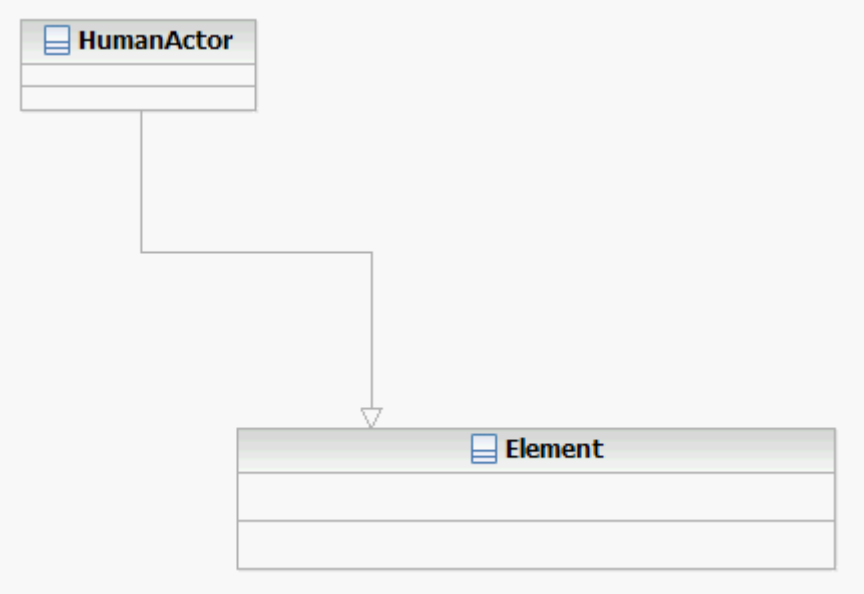

Figure 6: The HumanActor Class

HumanActor is defined as disjoint with the **Service** and **Task** classes. Instances of these classes are considered not to be people or organizations. **HumanActor** is specifically not defined as disjoint with **System** since an organization in many cases is in fact just a particular kind of system. We choose not to define a special intersection class to represent this fact.

3.3 HumanActor – Examples

3.3.1 The uses and usedBy Properties Applied to HumanActor

In one direction, a human actor can itself use things such as services, systems, and other human actors. In the other direction, a human actor can, for instance, be used by another actor or by a system (as an element within that system such as a human actor in a process).

3.3.2 The represents and representedBy Properties Applied to HumanActor

As mentioned in the introduction to this section, human actors are intrinsically part of systems that instantiate SOAs. Yet in many cases as an element of an SOA system we talk about not the specific person or organization, rather an abstract representation of them that participates in processes, provides services, etc. In other words, we talk about elements representing human actors.

As examples, a broker (instance of **HumanActor**) may represent a seller (instance of **HumanActor**) that wishes to remain anonymous, a role (instance of **Element**) may represent (the role aspect of) multiple instances of **HumanActor**, and an organizational unit (instance of **HumanActor**) may represent the many people (all instances of **HumanActor**) that are part of it.

Note that we have chosen not to define a "role class", as we believe that using **Element** with the **represents** property is a more general approach which does not limit the ability to also define role-based systems. For all practical purposes there is simply a "role subclass" of **Element**, a subclass that we have chosen not to define explicitly.

3.3.3 Organizational Example

Continuing the organizational example from above, we can now express that P1 (**John**), P2 (**Jack**), P3 (**Joe**), and P4 (**Mary**) as instances of **Element** are in fact (people) instances of **HumanActor**. We can also express (if we so choose) that all of O1 (**CarWashBusiness**), O3 (**CarWash**), and O4 (**Administration**) are (organization) human actors from an action perspective at the same time that they are systems from a collection/composition perspective.

3.3.4 Car Wash Example

See Section 8.1 for the complete organizational aspect of the car wash example.

3.4 The Task Class

```
<owl:Class rdf:about="#Task">
  <owl:disjointWith>
    <owl:Class rdf:about="#System"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#HumanActor"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Service"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Element"/>
  </rdfs:subClassOf>
</owl:Class>
```

A *task* is an atomic action which accomplishes a defined result. Tasks are done by people or organizations, specifically by instances of **HumanActor**.

The Business Process Modeling Notation (BPMN) 2.0 defines *task* as follows: "A *task* is an atomic Activity within a Process flow. A task is used when the work in the process cannot be broken down to a finer level of detail. Generally, an end-user and/or applications are used to perform the task when it is executed." For the purposes of the ontology we have added precision by formally separating the notion of doing from the notion of performing. Tasks are (optionally) done by human actors, furthermore (as instances of **Element**) tasks can use services that are performed by technology components (see details in Section 4.3; see also the example in Chapter 9).

The concept of *task* is captured by the **Task** OWL class, which is illustrated below (in Figure 7).

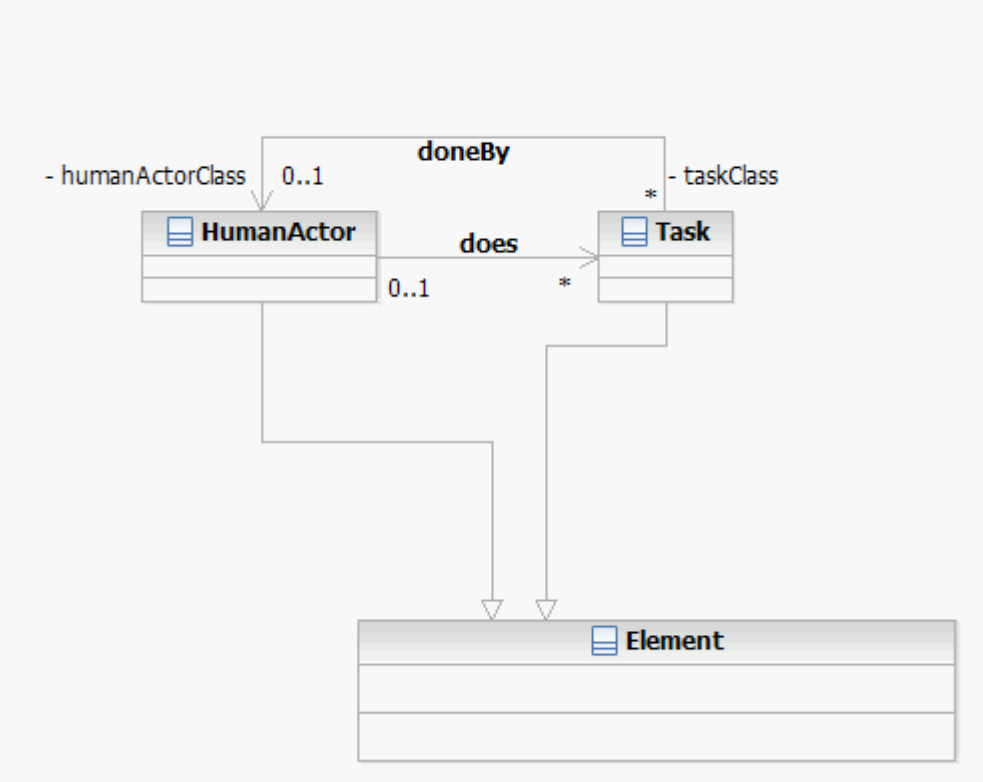

Figure 7: The Task Class

Task is defined as disjoint with the **System**, **Service**, and **HumanActor** classes. Instances of these classes are considered not to be atomic actions.

3.5 The does and doneBy Properties

```
<owl:ObjectProperty rdf:about="#doneBy">
  <rdfs:domain rdf:resource="#Task"/>
  <rdfs:range rdf:resource="#HumanActor"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#does">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#doneBy"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:Class rdf:about="#Task">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#doneBy"/>
```

```
      </owl:onProperty>
      <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >0</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:maxCardinality>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#doneBy"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Tasks are naturally thought of as being done by people or organizations. If we think of tasks as being the actual things done, then the natural cardinality is that each instance of **Task** is done by at most one instance of **HumanActor**. Due to the atomic nature of instances of **Task** we rule out the case where such an instance is done jointly by multiple instances of **HumanActor**. The cardinality can be zero if someone chooses not to instantiate all possible human actors. On the other hand, the same instance of **HumanActor** can (over time) easily do more than one instance of **Task**. The **does** property, and its inverse **doneBy**, capture the relation between a human actor and the tasks it performs.

3.6 Task – Examples

3.6.1 The uses and usedBy Properties Applied to Task

In one direction, the most common case of a task using another element is where an automated task (in an orchestrated process; see Chapter 5 for the definition of *process* and *orchestration*) uses a service as its realization. In the other direction, a task can, for instance, be used by a system (as an element within that system, such as a task in a process).

3.6.2 The represents and representedBy Properties Applied to Task

As mentioned in the introduction to this section, tasks are intrinsically part of SOA systems. Yet in many cases as an element of an SOA system we talk about not the actual thing being done, rather an abstract representation of it that is used as an element in systems, processes, etc. In other words, we talk about elements representing tasks.

As a simple example, an abstract activity in a process model (associated with a role) may represent a concrete task (done by a person fulfilling that role). Note that due to the atomic nature of a task it does not make sense to talk about many elements representing different aspects of it.

3.6.3 Organizational Example

Continuing the organizational example from above, we can now express which tasks that are done by human actors (people) P1, P2, P3, and P4, and how those tasks can be elements in bigger systems that describe things such as organizational processes. Chapter 5 will deal formally with the concept of *composition*, including properly defining the concept of a *process* as one particular kind of *composition*.

3.6.4 Car Wash Example

As an important part of the car wash system, John and Jack perform certain manual tasks required for washing a car properly:

- **Jack** and **John** are instances of **HumanActor**.
- **WashWindows** is an instance of **Task** and is done by **John**.
- **PushWashButton** is an instance of **Task** and is done by **Jack**.

4 Service, ServiceContract, and ServiceInterface

4.1 Introduction

Service is another core concept of this ontology. It is a concept that is fundamental to SOA and always used in practice when describing or engineering SOA systems, yet it is not easy to define formally. The ontology is based on the following definition of *service*:

"A service is a logical representation of a repeatable activity that has a specified outcome. It is self-contained and is a 'black box' to its consumers."

This corresponds to the existing official Open Group definition of the term; refer to the Open Group Definition of SOA.

The word "activity" in the definition above is here used in the general English language sense of the word, not in the process-specific sense of that same word (i.e., activities are not necessarily process activities). The ontology purposefully omits "business" as an intrinsic part of the definition of *service*. The reason for this is that the notion of business is relative to a person's viewpoint – as an example, one person's notion of IT is another person's notion of business (the business of IT). *Service* as defined by the ontology is agnostic to whether the concept is applied to the classical notion of a business domain or the classical notion of an IT domain.

Other current SOA-specific definitions of the term service include:

- *"A mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description."* (Source: OASIS SOA Reference Model)
- *"A capability offered by one entity or entities to others using well-defined 'terms and conditions' and interfaces."* (Source: OMG SoaML Specification)

Within the normal degree of precision of the English language, these definitions are not contradictory; they are stressing different aspects of the same concept. All three definitions are SOA-specific though, and represent a particular interpretation of the generic English language term service.

This chapter describes the following classes of the ontology:

- **Service**
- **ServiceContract**
- **ServiceInterface**
- **InformationType**

In addition, it defines the following properties:

- **performs** and **performedBy**
- **hasContract** and **isContractFor**
- **involvesParty** and **isPartyTo**
- **specifies** and **isSpecifiedBy**
- **hasInterface** and **isInterfaceOf**
- **hasInput** and **isInputAt**
- **hasOutput** and **isOutputAt**

4.2 The Service Class

```
<owl:Class rdf:about="#Service">
  <owl:disjointWith>
    <owl:Class rdf:about="#System"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Task"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#HumanActor"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Element"/>
  </rdfs:subClassOf>
</owl:Class>
```

A *service* is a logical representation of a repeatable activity that has a specified outcome. It is self-contained and is a 'black box' to its consumers. The concept of *service* is captured by the **Service** OWL class, which is illustrated below (in Figure 8).

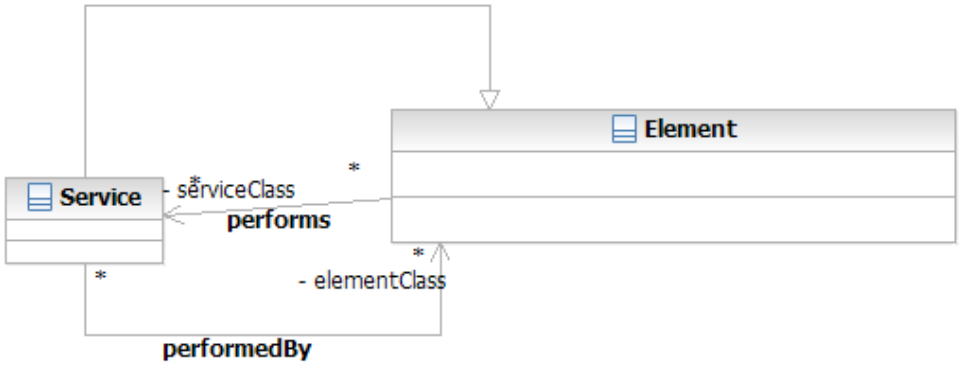

Figure 8: The Service Class

In the context of the SOA ontology we consider only SOA-based services. Other domains, such as Integrated Service Management, can have services that are not SOA-based and hence are outside the intended scope of the SOA ontology.

Service is defined as disjoint with the **System**, **Task**, and **HumanActor** classes. Instances of these classes are considered not to be services themselves, even though they may provide capabilities that can be offered as services.

4.3 The performs and performedBy Properties

```
<owl:ObjectProperty rdf:about="#performs">
  <rdfs:domain rdf:resource="#Element"/>
  <rdfs:range rdf:resource="#Service"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#performedBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#performs"/>
  </owl:inverseOf>
</owl:ObjectProperty>
```

As a service itself is only a logical representation, any service is *performed* by something. The something that *performs* a service must be opaque to anyone interacting with it, an opaqueness which is the exact nature of the **Element** class. This concept is captured by the **performs** and **performedBy** properties as illustrated in The Service Class (Figure 8). This also captures the fact that services can be performed by elements of other types than systems. This includes elements such as software components, human actors, and tasks.

Note that the same instance of **Service** can be performed by many different instances of **Element**. As long as the service performed is the same, an external observer cannot tell the difference (for contractual obligations, SLAs, etc. see the definition of the **ServiceContract** class in Section 4.5.). Conversely, any instance of **Element** may perform more than one service or none at all.

While a service can be performed by other elements, the service itself (as a purely logical representation) does not perform other services. See the Simple Service Composition Example (Section 5.6.1) for an example of how to represent service compositions formally in the ontology.

4.3.1 Service Consumers and Service Providers

Terminology used in an SOA environment often includes the notion of service providers and service consumers. There are two challenges with this terminology:

- It does not distinguish between the contractual obligation aspect of consume/provide and the interaction aspect of consume/provide. A contractual obligation does not necessarily translate to an interaction dependency, if for no other reason than because the realization of the contractual obligation may have been sourced to a third party.

- Consuming or providing a service is a statement that only makes sense in context – either a contractual context or an interaction context. These terms are consequently not well suited for making statements about elements and services in isolation.

The above are the reasons why the ontology has chosen not to adopt consume and provide as core concepts, rather instead allows consume or provide terms used with contractual obligations and/or interaction rules described by service contracts; see the definition of the **ServiceContract** class in Section 4.5. In its simplest form, outside the context of a formal service contract, the interaction aspect of consuming and providing services may even be expressed simply by saying that some element uses (consumes) a service or that some element performs (provides) a service; see also the examples below.

4.4 Service – Examples

4.4.1 The uses and usedBy Properties Applied to Service

In one direction, it does not really make sense to talk about a service that uses another element. While the thing that performs the service might very well include the use of other elements (and certainly will in the case of *service composition*), the service itself (as a purely logical representation) does not use other elements.

In the other direction, we find the most common of all interactions in an SOA environment: the notion that some element uses a service by interacting with it. Note that from an operational perspective this interaction actually reaches somewhat beyond the service itself by involving the following typical steps:

- Picking the service to interact with (this statement is agnostic as to whether this is done dynamically at runtime or statically at design and/or construct time)
- Picking an element that performs that service (in a typical SOA environment, this is most often done "inside" an Enterprise Service Bus (ESB))
- Interacting with the chosen element (that performs the chosen) service (often also facilitated by an ESB)

4.4.2 The represents and representedBy Properties Applied to Service

Concepts such as service mediations, service proxies, ESBs, etc. are natural to those practitioners that describe and implement the operational aspects of SOA systems. From an ontology perspective all of these can be captured by some other element representing the service – a level of indirection that is critical when we do not want to bind operationally to a particular service endpoint, rather we want to preserve loose-coupling and the ability to switch embodiments as needed. Note that by leveraging the **represents** and **representedBy** properties in this fashion we additionally encapsulate the relatively complex operational interaction pattern that was described in the section above (picking the service, picking an element that performs the service, and interacting with that chosen element).

While a service being represented by something else is quite natural, it is harder to imagine what the service itself might represent. To some degree we have already captured the fact that a service represents any embodiment of it, only we have chosen to use the **performs** and **performedBy** properties to describe this rather than the generic **represents** and **representedBy** properties. As a consequence, we do not expect practical applications of the ontology to have services represent anything.

4.4.3 Exemplifying the Difference between Doing a Task and Performing a Service

The distinction between a human actor doing a task and an element (technology, human actor, or other) performing a service is important. The human actor doing the task has the responsibility that it gets done, yet may in fact in many cases leverage some service to achieve that outcome:

- **John** is an instance of **HumanActor**.
- **WashWindows** is an instance of **Task** and is done by **John**.
- **SoapWater** is an instance of **Service**.
- **WaterTap** is an instance of **Element**.
- **WaterTap** performs **SoapWater**.
- **John** uses **SoapWater** (to do **WashWindows**).

Note how clearly **SoapWater** does not do **WashWindows**, nor does **WaterTap** do **WashWindows**.

4.4.4 Car Wash Example

Joe offers two different services to his customers: a basic wash and a gold wash. This can be instantiated in the ontology in the following way (subset to the part relevant for these two services):

- **GoldWash** is an instance of **Service**.
- **BasicWash** is an instance of **Service**.
- **CarWash** performs both **BasicWash** and **GoldWash**.
- **WashManager** represents both **BasicWash** and **GoldWash** (i.e., is the interaction point where customers can order services as well as pay for them).

Note the purposeful use of **WashManager** representing both services. This is due to Joe deciding that in his car wash customers are not to interact with the washing machinery directly, rather must instead interact with whomever (human actor) is fulfilling the role of wash manager.

4.5 The ServiceContract Class

```
<owl:Class rdf:about="#ServiceContract">
  <owl:disjointWith>
    <owl:Class rdf:about="#HumanActor"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Task"/>
  </owl:disjointWith>
</owl:Class>
```

In many cases, specific agreements are needed in order to define how to use a service. This can either be because of a desire to regulate such use or can simply be because the service will not function properly unless interaction with it is done in a certain sequence. A *service contract* defines the terms, conditions, and interaction rules that interacting participants must agree to (directly or indirectly). A service contract is binding on all participants in the interaction, including the service itself and the element that provides it for the particular interaction in question. The concept of *service contract* is captured by the **ServiceContract** OWL class, which is illustrated below (in Figure 9).

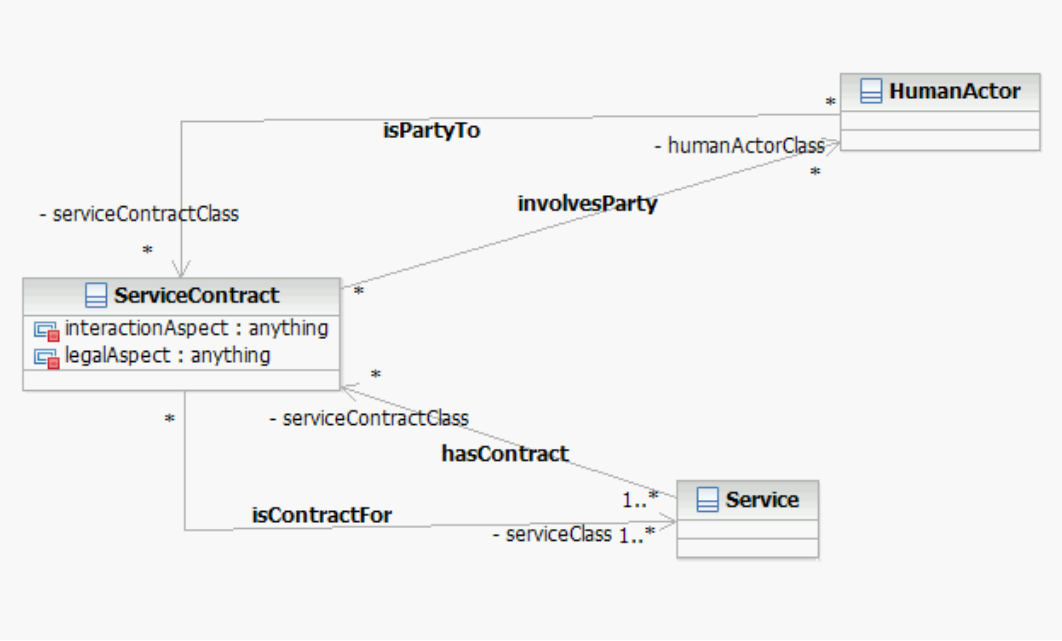

Figure 9: The ServiceContract Class

4.5.1 The interactionAspect and legalAspect Datatype Properties

```
<owl:DatatypeProperty rdf:about="#interactionAspect">
  <rdfs:domain rdf:resource="#ServiceContract"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:about="#legalAspect">
  <rdfs:domain rdf:resource="#ServiceContract"/>
```

```
</owl:DatatypeProperty>

<owl:Class rdf:about="#ServiceContract">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="#legalAspect"/>
      </owl:onProperty>
      <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:maxCardinality>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="#legalAspect"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="#interactionAspect"/>
      </owl:onProperty>
      <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="#interactionAspect"/>
      </owl:onProperty>
      <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Service contracts explicitly regulate both the interaction aspects (see the **hasContract** and **isContractFor** properties) and the legal agreement aspects (see the **involvedParty** and **isPartyTo** properties) of using a service. The two types of aspects are formally captured by defining the **interactionAspect** and **legalAspect** datatype properties on the **ServiceContract** class. Note that the second of these attributes, the legal agreement aspects, includes concepts such as Service-Level Agreements (SLAs).

If desired, it is possible as an architectural convention to split the interaction and legal aspects into two different service contracts. Such choices will be up to any application using this ontology.

4.6 The hasContract and isContractFor Properties

```
<owl:ObjectProperty rdf:about="#isContractFor">
  <rdfs:domain rdf:resource="#ServiceContract"/>
  <rdfs:range rdf:resource="#Service"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasContract">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#isContractFor"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:Class rdf:about="#ServiceContract">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#isContractFor"/>
      </owl:onProperty>
      <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

The **hasContract** property, and its inverse **isContractFor**, capture the abstract notion of a service having a service contract. Anyone wanting to use a service must obey the interaction aspects (as defined in the **interactionAspect** datatype property) of any service contract applying to that interaction. In that fashion, the interaction aspects of a service contract are context-independent; they capture the defined or intrinsic ways in which a service may be used.

By definition, any service contract must be a contract for at least one service. It is possible that the same service contract can be a contract for more than one service; for instance, in cases where a group of services share the same interaction pattern or where a service contract (legally – see the **involvesParty** and **isPartyTo** properties below) regulates the providing and consuming of multiple services.

4.7 The involvesParty and isPartyTo Properties

```
<owl:ObjectProperty rdf:about="#isPartyTo">
  <rdfs:domain rdf:resource="#HumanActor"/>
  <rdfs:range rdf:resource="#ServiceContract"/>
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:about="#involvesParty">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#isPartyTo"/>
  </owl:inverseOf>
</owl:ObjectProperty/>
```

In addition to the rules and regulations that intrinsically apply to any interaction with a service (the interaction aspect of service contracts captured in the **interactionAspect** datatype property) there may be additional legal agreements that apply to certain human actors and their use of services. The **involvesParty** property, and its inverse **isPartyTo**, capture the abstract notion of a service contract specifying legal obligations between human actors in the context of using the one or more services for which the service contract is a contract.

While the **involvesParty** and **isPartyTo** properties define the relationships to human actors involved in the service contract, the actual legal obligations on each of these human actors is defined in the **legalAspect** datatype property on the service contract. This includes the ability to define who is the provider and who is the consumer from a legal obligation perspective.

There is a many-to-many relationship between service contracts and human actors. A given human actor may be party to none, one, or many service contracts. Similarly, a given service contract may involve none, one, or multiple human actors (none in the case where that particular service contract only specifies the **interactionAspect** datatype property). Note that it is important we allow for sourcing contracts where there is a legal agreement between human actor A and human actor B (both of which are party to a service contract), yet human actor B has sourced the performing of the service to human actor C (*aka* human actor C performs the service in question, not human actor B).

The **involvesParty** property together with the **legalAspect** datatype property on **ServiceContract** capture not just transient obligations. They include the ability to express “is obliged to at this instant”, “was obliged to”, and “may in future be obliged to”.

4.8 The Effect Class

```
<owl:Class rdf:about="#Effect">
  <owl:disjointWith>
    <owl:Class rdf:about="#ServiceInterface"/>
  </owl:disjointWith>
</owl:Class>
```

Interacting with something performing a service has *effects*. These comprise the outcome of that interaction, and are how a service (through the element that performs it) delivers value to its consumers. The concept of *effect* is captured by the **Effect** OWL class, which is illustrated below (in Figure 10).

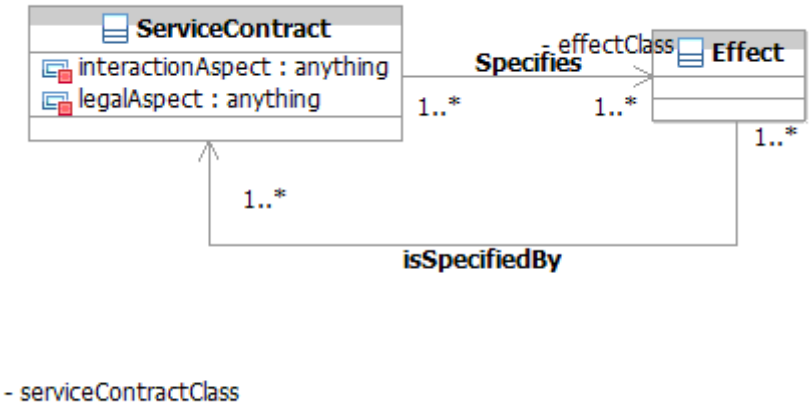

Figure 10: The Effect Class

Note that the **Effect** class purely represents how results or value is delivered to someone interacting with a service. Any possible internal side-effects are explicitly not covered by the **Effect** class.

Effect is defined as disjoint with the **ServiceInterface** class. (The **ServiceInterface** class is defined later in this document.) Interacting with a service through its service interface can have an outcome or provide a value (an instance of **Effect**), but the service interface itself does not constitute that outcome or value.

4.9 The specifies and isSpecifiedBy Properties

```
<owl:ObjectProperty rdf:about="#specifies">
  <rdfs:domain rdf:resource="#ServiceContract"/>
  <rdfs:range rdf:resource="#Effect"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#isSpecifiedBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#specifies"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:Class rdf:about="#Effect">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#isSpecifiedBy"/>
      </owl:onProperty>
    </owl:Restriction>
```

```
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="#ServiceContract">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#specifies"/>
      </owl:onProperty>
      <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

While a service intrinsically has an effect every time someone interacts with it, in order to trust the effect to be something in particular, the effect needs to be specified as part of a service contract. The **specifies** property, and its inverse **isSpecifiedBy**, capture the abstract notion of a service contract specifying a particular effect as part of the agreement for using a service. Note that the specified effect can apply to both the **interactionAspect** datatype property (simply specifying what will happen when interacting with the service according to the service contract) and the **legalAspect** datatype property (specifying a contractually promised effect).

Anyone wanting a guaranteed effect of the interaction with a given service must ensure that the desired effect is specified in a service contract applying to that interaction. By definition, any service contract must specify at least one effect. In the other direction, an effect must be an effect of at least one service contract; this represents that fact that we have chosen only to formalize those effects that are specified by service contracts (and not all intrinsic effects of all services).

4.10 ServiceContract – Examples

4.10.1 Service-Level Agreements

A Service-Level Agreement (SLA) on a service has been agreed by organizations **A** and **B**. It is important to realize that an SLA always has a context of the parties that have agreed to it, involving at a minimum one legal “consumer” and one legal “provider”. This can be represented in the ontology as follows:

- **A** and **B** are instances of **HumanActor**.
- **Service** is an instance of **Service**.
- **ServiceContract** is an instance of **ServiceContract**.
- **ServiceContract isContractFor Service**.
- **ServiceContract involvesParty A**.
- **ServiceContract involvesParty B**.

The **legalAspect** datatype property on **ServiceContract** describes the SLA.

4.10.2 Service Sourcing

Organizations **A** and **B** have agreed on **B** providing certain services for **A**, yet **B** wants to source the actual delivery of those services to third-party **C**. This can be represented in the ontology as follows:

- **A**, **B**, and **C** are instances of **HumanActor**.
- **Service** is an instance of **Service**.
- **C** provides **Service**.
- **ServiceContract** is an instance of **ServiceContract**.
- **ServiceContract isContractFor Service**.
- **ServiceContract involvesParty A**.
- **ServiceContract involvesParty B**.

The **legalAspect** datatype property on **ServiceContract** describes the legal obligation of **B** to provide **Service** for **A**.

4.10.3 Car Wash Example

See Section 8.2 for the complete **Service** and **ServiceContract** aspects of the car wash example.

4.11 The ServiceInterface Class

```
<owl:Class rdf:about="#ServiceInterface">
  <owl:disjointWith>
    <owl:Class rdf:about="#Service"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#ServiceContract"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Effect"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#HumanActor"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Task"/>
  </owl:disjointWith>
</owl:Class>
```

An important characteristic of services is that they have simple, well-defined interfaces. This makes it easy to interact with them, and enables other elements to use them in a structured

manner. A *service interface* defines the way in which other elements can interact and exchange information with a service. This concept is captured by the **ServiceInterface** OWL class which is illustrated below (in Figure 11).

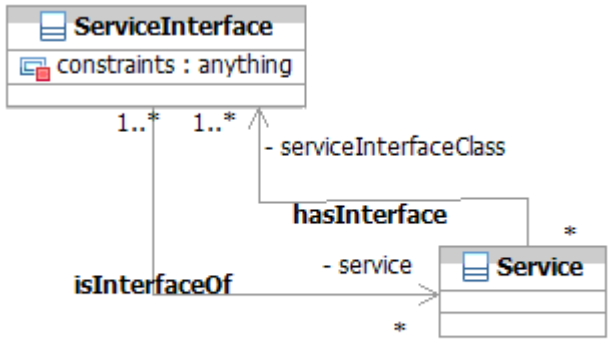

Figure 11: The ServiceInterface Class

The concept of an interface is in general well understood by practitioners, including the notion that interfaces define the parameters for information passing in and out of them when invoked. What differs from domain to domain is the specific nature of how an interface is invoked and how information is passed back and forth. Service interfaces are typically, but not necessarily, message-based (to support loose-coupling). Furthermore, service interfaces are always defined independently from any service implementing them (to support loose-coupling and service mediation).

From a design perspective interfaces may have more granular operations or may be composed of other interfaces. We have chosen to stay at the concept level and not include such design aspects in the ontology.

ServiceInterface is defined as disjoint with the **Service**, **ServiceContract**, and **Effect** classes. Instances of these classes are considered not to define (by themselves) the way in which other elements can interact and exchange information with a service. Note that there is a natural synergy between **ServiceInterface** and the **interactionAspect** datatype property on **ServiceContract**, as the latter defines any multi-interaction and/or sequencing constraints on how to use a service through interaction with its service interfaces.

4.11.1 The Constraints Datatype Property

```
<owl:DatatypeProperty rdf:about="#constraints">
  <rdfs:domain rdf:resource="#ServiceInterface"/>
</owl:DatatypeProperty>

<owl:Class rdf:about="#ServiceInterface">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="#constraints"/>
```

```
      </owl:onProperty>
      <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="#constraints"/>
      </owl:onProperty>
      <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

The **Constraints** datatype property on **ServiceInterface** captures the notion that there can be constraints on the allowed interaction such as only certain value ranges allowed on given parameters. Depending on the nature of the service and the service interface in question, these constraints may be defined either formally or informally (the informal case being relevant at a minimum for certain types of real-world services).

4.12 The hasInterface and isInterfaceOf Properties

```
<owl:ObjectProperty rdf:about="#hasInterface">
  <rdfs:domain rdf:resource="#Service"/>
  <rdfs:range rdf:resource="#ServiceInterface"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#isInterfaceOf"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#hasInterface"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:Class rdf:about="#Service">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasInterface"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

The **hasInterface** property, and its inverse **isInterfaceOf**, capture the abstract notion of a service having a particular service interface.

In one direction, any service must have at least one service interface; anything else would be contrary to the definition of a service as a representation of a repeatable activity that has a specified outcome and is a 'black box' to its consumers. In the other direction, there can be service interfaces that are not yet interfaces of any defined services. Also, the same service interface can be an interface of multiple services. The latter does not mean that these services are the same, nor even that they have the same effect; it only means that it is possible to interact with all these services in the manner defined by the service interface in question.

4.13 The InformationType Class

```
<owl:Class rdf:about="#InformationType">
  <owl:disjointWith>
    <owl:Class rdf:about="#Effect"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#ServiceContract"/>
  </owl:disjointWith>
</owl:Class>
```

A service interface can enable another element to give information to or receive information from a service (when it uses that service); specifically the types of information given or received. The concept of *information type* is captured by the **InformationType** OWL class, which is illustrated below (in Figure 12).

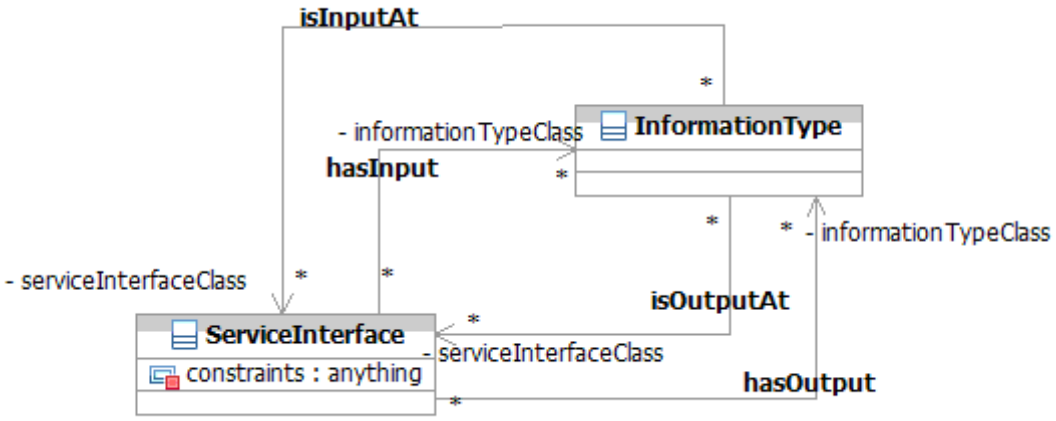

Figure 12: The InformationType Class

In any concrete interaction through a service interface the information types on that interface are instantiated by information items, yet for the service interface itself it is the types that are important. Note that the **constraints** datatype property on **ServiceInterface**, if necessary, can be used to express constraints on allowed values for certain information types.

4.14 The hasInput and isInputAt Properties

```
<owl:ObjectProperty rdf:about="#hasInput">
  <rdfs:domain rdf:resource="#ServiceInterface"/>
  <rdfs:range rdf:resource="#InformationType"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#isInputAt">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#hasInput"/>
  </owl:inverseOf>
</owl:ObjectProperty>
```

The **hasInput** property, and its inverse **isInputAt**, capture the abstract notion of a particular type of information being given when interacting with a service through a service interface.

Note that there is a many-to-many relationship between service interfaces and input information types. A given information type may be input at many service interfaces or none at all. Similarly, a given service interface may have many information types as input or none at all. It is important to realize that some services may have only inputs (triggering an asynchronous action without a defined response) and other services may have only outputs (elements performing these services execute independently yet may provide output that is used by other elements).

4.15 The hasOutput and isOutputAt Properties

```
<owl:ObjectProperty rdf:about="#hasOutput">
  <rdfs:domain rdf:resource="#ServiceInterface"/>
  <rdfs:range rdf:resource="#InformationType"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#isOutputAt">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#hasOutput"/>
  </owl:inverseOf>
</owl:ObjectProperty>
```

The **hasOutput** property, and its inverse **isOutputAt**, capture the abstract notion of a particular type of information being received when interacting with a service through a service interface.

Note that there is a many-to-many relationship between service interfaces and output information types. A given information type may be output at many service interfaces or none at all. Similarly, a given service interface may have many information types as output or none at all. It is important to realize that some services may have only inputs (triggering an asynchronous action without a defined response) and other services may have only outputs (elements performing these services execute independently yet may provide output that is used by other elements).

4.16 Examples

4.16.1 Interaction Sequencing

A service contract on a service expresses that the services interfaces on that service must be used in a certain order:

- **Service** is an instance of **Service**.
- **ServiceContract** is an instance of **ServiceContract**.
- **ServiceContract isContractFor Service**.
- **X** is an instance of **ServiceInterface**.
- **X isInterfaceOf Service**.
- **Y** is an instance of **ServiceInterface**.
- **Y isInterfaceOf Service**.

The **interactionAspect** datatype property on **ServiceContract** describes that **X** must be used before **Y** may be used.

4.16.2 Car Wash Example

See Section 8.2 for the complete **ServiceInterface** aspect of the car wash example.

5 Composition and its Subclasses

5.1 Introduction

The notion of *composition* is a core concept of SOA. Services can be composed of other services. Processes are composed of human actors, tasks, and possibly services. Experienced SOA practitioners intuitively apply composition as an integral part of architecting, designing, and realizing SOA systems; in fact, any well structured SOA environment is intrinsically composite in the way services and processes support business capabilities. What differs from practitioner to practitioner is the exact nature of the composition – the composition pattern being applied.

This chapter describes the following classes of the ontology:

- **Composition** (as a subclass of **System**)
- **ServiceComposition** (as a subclass of **Composition**)
- **Process** (as a subclass of **Composition**)

In addition, it defines the following datatype property:

- **compositionPattern**

5.2 The Composition Class

```
<owl:Class rdf:about="#Composition">
  <rdfs:subClassOf>
    <owl:Class rdf:about="#System"/>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl: rdf:about="#Task"/>
  </owl:disjointWith>
</owl:Class>
```

A *composition* is the result of assembling a collection of things for a particular purpose. Note in particular that we have purposefully distinguished between the act of composing and the resulting composition as a thing, and that it is in the latter sense we are using the concept of composition here. The concept of *composition* is captured by the **Composition** OWL class, which is illustrated below (in Figure 13).

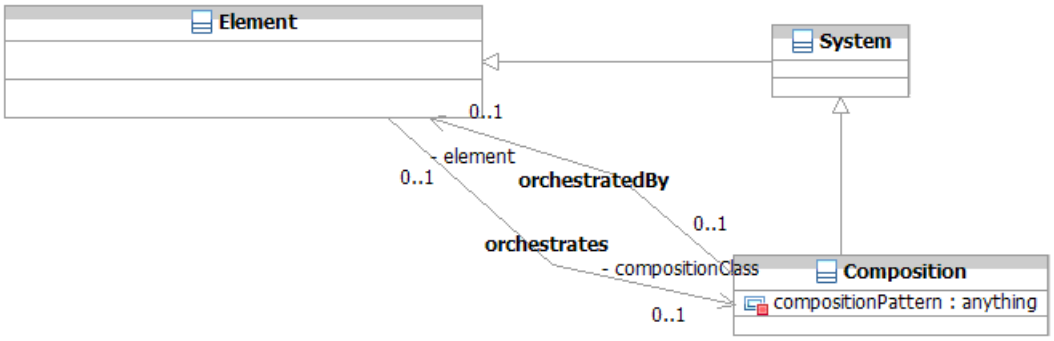

Figure 13: The Composition Class

Being intrinsically (also) an organized collection of other, simpler things, the **Composition** class is a subclass of the **System** class. While a composition is always also a system, a system is not necessarily a composition in that it is not necessarily a result of anything – note here the difference between a system producing a result and the system itself being a result. A perhaps more tangible difference between a system and a composition is that the latter must have associated with it a specific composition pattern that renders the composition (as a whole) as the result when that composition pattern is applied to the elements used in the composition. One implication of this is that there is not a single member of a composition that represents (as an element) that composition as a whole; in other words, the composition itself is not one of the things being assembled. On the other hand, composition is in fact a recursive concept (as are all subclasses of **System**) – being a system, a composition is also an element which means that it can be used by a higher-level composition.

In the context of the SOA ontology we consider in detail only functional compositions that belong to the SOA domain. Note that a fully described instance of **Composition** must have by its nature a *uses* relationship to at least one instance of **Element.** (It need not necessarily have more than one as the composition pattern applied may be, for instance, simply a transformation.) Again (as for **System**) it is important to realize that a composition can use elements outside its own boundary.

Since **Composition** is a subclass of **Element**, all compositions have a boundary and are opaque to an external observer (black box view). The composition pattern in turn is the internal viewpoint (white box view) of a composition. As an example, for the notion of a service composition this would correspond to the difference between seeing the service composition as an element providing a (higher-level) service or seeing the service composition as a composite structure of (lower-level) services.

5.2.1 The compositionPattern Datatype Property

```
<owl:DatatypeProperty rdf:about="#compositionPattern">
    <rdfs:domain rdf:resource="#Composition"/>
</owl:DatatypeProperty>

<owl:Class rdf:about="#Composition">
```

```
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:maxCardinality>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="#compositionPattern"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="#compositionPattern"/>
      </owl:onProperty>
      <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

As discussed above, any composition must have associated with it a specific composition pattern, that pattern describing the way in which a collection of elements is assembled to a result. The concept of a composition pattern is captured by the **compositionPattern** datatype property. Note that even though certain kinds of composition patterns are of special interest within SOA (see below), the **compositionPattern** datatype property may take any value as long as that value describes how to assemble the elements used by the composition with which it is associated.

The Orchestration Composition Pattern

One kind of composition pattern that has special interest within SOA is an *orchestration*. In an orchestration (a composition whose composition pattern is an orchestration), there is one particular element used by the composition that oversees and directs the other elements. Note that the element that directs an orchestration by definition is different than the orchestration (**Composition** instance) itself.

Think of an orchestrated executable workflow as an example of an orchestration. The workflow construct itself is one of the elements being used in the composition, yet it is different from the composition itself – the composition itself is the result of applying (executing) the workflow on the processes, human actors, services, etc. that are orchestrated by the workflow construct.

A non-IT example is the foreman of a road repair crew. If the foreman chooses to exert direct control over the tasks done by his crew, than the resulting composition becomes an orchestration (with the foreman as the director and provider of the composition pattern). Note that under other circumstances, with a different team composition model, a road repair crew can also act as a collaboration or a choreography. (See below for definitions of *collaboration* and *choreography*.)

As the last example clearly shows, using an orchestration composition pattern is not a guarantee that "nothing can go wrong". That would, in fact, depend on the orchestration director's ability to handle exceptions.

The Choreography Composition Pattern

Another kind of composition pattern that has special interest within SOA is a *choreography*. In a choreography (a composition whose composition pattern is a choreography) the elements used by the composition interact in a non-directed fashion, yet with each autonomous member knowing and following a predefined pattern of behavior for the entire composition.

Think of a process model as an example of a choreography. The process model does not direct the elements within it, yet does provide a predefined pattern of behavior that each such element is expected to conform to when "executing".

The Collaboration Composition Pattern

A third kind of composition pattern that has special interest within SOA is a *collaboration*. In a collaboration (a composition whose composition pattern is a collaboration) the elements used by the composition interact in a non-directed fashion, each according to their own plans and purposes without a predefined pattern of behavior. Each element simply knows what it has to do and does it independently, initiating interaction with the other members of the composition as applicable on its own initiative. This means that there is no overall predefined "flow" of the collaboration, though there may be a run-time "observed flow of interactions".

A good example of a collaboration is a work meeting. There is no script for how the meeting will unfold and only after the meeting has concluded can we describe the sequence of interactions that actually occurred.

5.3 The orchestrates and orchestratedBy Properties

```
<owl:ObjectProperty rdf:about="#orchestratedBy">
  <rdfs:domain rdf:resource="#Composition"/>
  <rdfs:range rdf:resource="#Element"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#orchestrates">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#orchestratedBy"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:Class rdf:about="#Composition">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:maxCardinality>
```

```
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#orchestratedBy"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >0</owl:minCardinality>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#orchestratedBy"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="#Element">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >0</owl:minCardinality>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#orchestrates"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:maxCardinality>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#orchestrates"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

As defined above, an orchestration has one particular element that oversees and directs the other elements used by the composition. This type of relationship is important enough that we have chosen to capture the abstract notion in the **orchestrates** property and its inverse **orchestratedBy**.

In one direction, a composition has at most one element that orchestrates it, and the cardinality can only be one (1) if in fact the composition pattern of that composition is an orchestration. In the other direction, an element can orchestrate at most one composition which then must have an orchestration as its composition pattern.

Note that in practical applications of the ontology, even though **Service** is a subclass of **Element**, a service (as a purely logical representation) is not expected to orchestrate a composition.

5.4 The ServiceComposition Class

```
<owl:Class rdf:about="#ServiceComposition">
  <rdfs:subClassOf>
    <owl: rdf:about="#Composition"/>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:about="#ServiceContract"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl: rdf:about="#ServiceInterface"/>
  </owl:disjointWith>
</owl:Class>
```

A key SOA concept is the notion of *service composition*, the result of assembling a collection of services in order to perform a new higher-level service. The concept of *service composition* is captured by the **ServiceComposition** OWL class, which is illustrated below (in Figure 14).

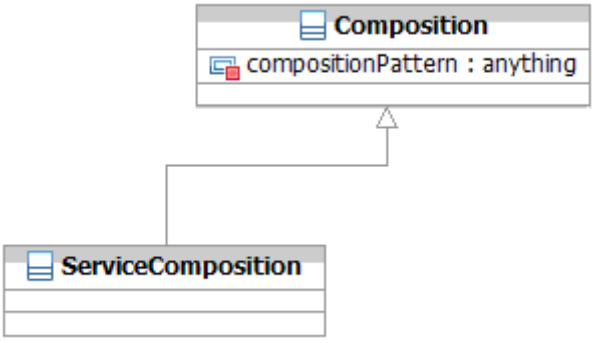

Figure 14: The ServiceComposition Class

As a service composition is the result of assembling a collection of services, **ServiceComposition** is naturally a subclass of **Composition**.

A service composition may, and typically will, add logic (or even "code") via the composition pattern. Note that a service composition is *not* the new higher-level service itself (due to the **System** and **Service** classes being disjoint); rather it performs (as an element) that higher-level service.

5.5 The Process Class

```
<owl:Class rdf:about="#Process">
  <rdfs:subClassOf>
```

```
    <owl:Class rdf:about="#Composition"/>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:about="#ServiceContract"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#ServiceInterface"/>
  </owl:disjointWith>
</owl:Class>
```

Another key SOA concept is the notion of *process*. A process is a composition whose elements are composed into a sequence or flow of activities and interactions with the objective of carrying out certain work. This definition is consistent with, for instance, the Business Process Modeling Notation (BPMN) 2.0 definition of a *process*. The concept of *process* is captured by the **Process** OWL class, which is illustrated below (in Figure 15).

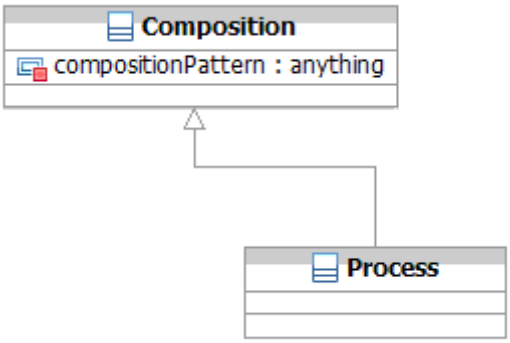

Figure 15: The Process Class

Elements in process compositions can be things like human actors, tasks, services, other processes, etc. A process always adds logic via the composition pattern; the result is more than the parts. According to their collaboration pattern, processes can be:

- **Orchestrated**: When a process is orchestrated in a business process management system, then the resulting IT artifact is in fact an orchestration; i.e., it has an orchestration collaboration pattern. This type of process is often called a *process orchestration*.
- **Choreographed**: For example, a process model representing a defined pattern of behavior. This type of process is often called a *process choreography*.
- **Collaborative**: No (pre)defined pattern of behavior (model); the process represents observed (executed) behavior.

5.6 Service Composition and Process Examples

5.6.1 Simple Service Composition Example

Using a service composition example, services **A** and **B** are instances of **Service** and the composition of **A** and **B** is an instance of **ServiceComposition** (that uses **A** and **B**):

- **A** and **B** are instances of **Service**.
- **X** is an instance of **ServiceComposition**.
- **X** uses both **A** and **B** (composes them according to its service composition pattern).

Note that there are various ways in which the service composition pattern can compose A and B, all of which are relevant in one situation or another. For example, interfaces of **X** may or may not include some subset of the interfaces of **A** and **B**. Furthermore, the interfaces of **A** and **B** may or may not also be (directly) invocable without going through **X** – that is a matter of the service contracts and/or access policies that apply to **A** and **B**. Finally, **X** may also use other elements that are not services at all (examples are composition code, adaptors, etc.).

5.6.2 Process Example

Using a process example, tasks **T1** and **T2** are instances of **Task**, roles **R1** and **R2** are instances of **Element**, and the composition of **T1**, **T2**, **R1**, and **R2** is an instance of **Process** (that uses **T1**, **T2**, **R1**, and **R2**):

- **T1** and **T2** are instances of **Task**.
- **R1** and **R2** are instances of **Element**.
- **Y** is an instance of **Process**.
- **Y** uses all of **T1**, **T2**, **R1**, and **R2** (composes them according to its process composition pattern).

5.6.3 Process and Service Composition Example

Elaborating on the process example above, if **T1** is done using service **S** then:

- **S** is an instance of **Service**.
- **T1** uses **S**.

Note that depending on the particular design approach chosen (and the resulting composition pattern), **Y** may or may not use **S** directly. This depends on whether **Y** carries the binding between **T1** and **S** or whether that binding is encapsulated in **T1**.

5.6.4 Car Wash Example

See Section 8.3 for the **Process** aspect of the car wash example.

6 Policy

6.1 Introduction

Policies, the human actors defining them, and the things that they apply to are important aspects of any system, certainly also SOA systems with their many different interacting elements. Policies can apply to any element in a system. The concept of *policy* is captured by the **Policy** class and its relationships to the **HumanActor** and **Thing** classes.

This chapter describes the following classes of the ontology:

- **Policy**

In addition, it defines the following properties:

- **appliesTo** and **isSubjectTo**
- **setsPolicy** and **isSetBy**

6.2 The Policy Class

```
<owl:Class rdf:about="#Policy">
  <owl:disjointWith>
    <owl:Class rdf:about="#InformationType"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#ServiceInterface"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Element"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Effect"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Event"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#ServiceContract"/>
  </owl:disjointWith>
</owl:Class>
```

A *policy* is a statement of direction that a human actor may intend to follow or may intend that another human actor should follow. Knowing the policies that apply to something makes it

easier and more transparent to interact with that something. The concept of *policy* is captured by the **Policy** OWL class, which is illustrated below (in Figure 16).

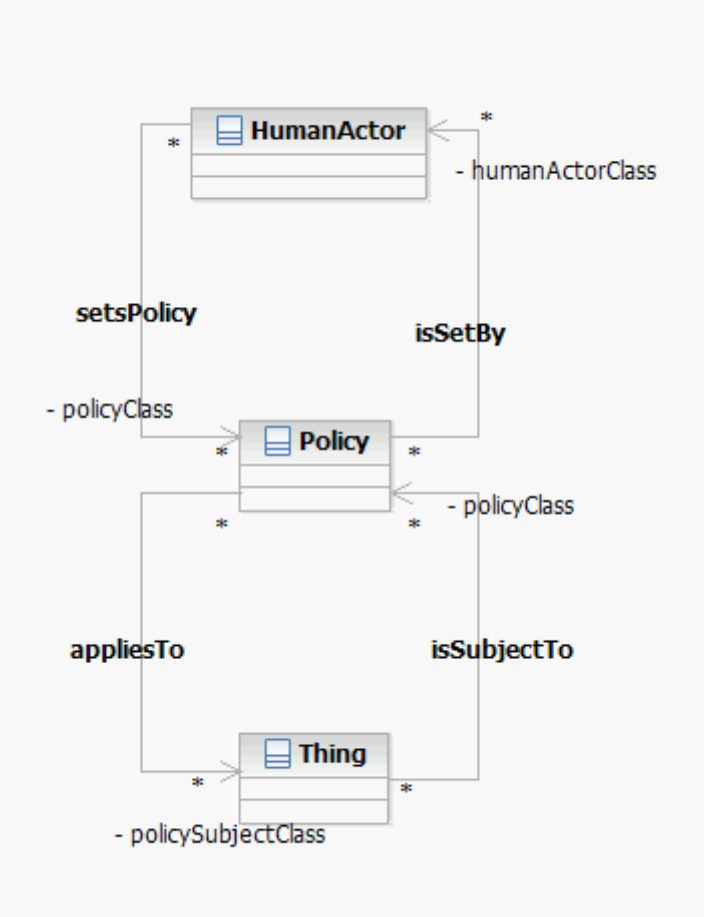

Figure 16: The Policy Class

Policy as a concept is generic and has relevance outside the domain of SOA. For the purposes of this SOA ontology it has not been necessary or relevant to restrict the generic nature of the **Policy** class itself. The relationships between **Policy** and **HumanActor** are of course bound by the SOA-specific restrictions that have been applied on the definition of **HumanActor**.

From a design perspective policies may have more granular parts or may be expressed and made operational through specific rules. We have chosen to stay at the concept level and not include such design aspects in the ontology.

Policy is distinct from all other concepts in this ontology, hence the **Policy** class is defined as disjoint with all other defined classes. In particular, **Policy** is disjoint with **ServiceContract**. While policies may apply to service contracts – such as security policies on who may change a given service contract – or conversely be referred to by service contracts as part of the terms, conditions, and interaction rules that interacting participants must agree to, service contracts are themselves not policies as they do not describe an intended course of action.

6.3 The appliesTo and isSubjectTo Properties

```
<owl:ObjectProperty rdf:about="#appliesTo">
  <rdfs:domain rdf:resource="#Policy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#isSubjectTo">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#appliesTo"/>
  </owl:inverseOf>
</owl:ObjectProperty>
```

Policies can apply to things other than elements; in fact, policies can apply to anything at all, including other policies. For instance, a security policy might specify which actors have the authority to change some other policy. The **appliesTo** property, and its inverse **isSubjectTo**, capture the abstract notion that a policy can apply to any instance of **Thing**. Note specifically that **Element** is a subclass of **Thing**, hence policies by inference can apply to any instance of **Element**.

In one direction, a policy can apply to zero (in the case where a policy has been formulated but not yet explicitly applied to anything), one, or more instances of **Thing**. Note that having a policy apply to multiple things does not mean that these things are the same, only that they are (partly) regulated by the same intent. In the other direction, an instance of **Thing** may be subject to zero, one, or more policies. Note that where multiple policies apply to the same instance of **Thing** this is often because the multiple policies are from multiple different policy domains (such as security and governance).

The SOA ontology does not attempt to enumerate different policy domains; such policy-focused details are deemed more appropriate for a policy ontology. It is worth pointing out that a particular policy ontology may also restrict (if desired) the kinds of things that policies can apply to.

6.4 The setsPolicy and isSetBy Properties

```
<owl:ObjectProperty rdf:about="#setsPolicy">
  <rdfs:domain rdf:resource="#HumanActor"/>
  <rdfs:range rdf:resource="#Policy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#isSetBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#setsPolicy"/>
  </owl:inverseOf>
</owl:ObjectProperty>
```

The **setsPolicy** property, and its inverse **isSetBy**, capture the abstract notion that a policy can be set by one or more human actors.

In one direction, a policy can be set by zero (in the case where actors setting the policy by choice are not defined or captured), one, or more human actors. Note specifically that some policies are set by multiple human actors in conjunction, meaning that all these human actors need to discuss and agree on the policy before it can take effect. A real-world example would be two parents in conjunction setting policies for acceptable child behavior. In the other direction, a human actor may potentially set (or be part of setting) multiple policies.

The SOA ontology purposefully separates the setting of the policy itself and the application of the policy to one or more instances of **Thing**. In some cases these two acts may be inseparably bound together, yet in other cases they are definitely not. One such example is an overall compliance policy that is formulated at the corporate level yet applied by the compliance officer in each line of business.

Also, while a particular case of interest for this ontology is that where the provider of a service has a policy for the service, a policy for a service is not necessarily owned by the provider. For example, government food and hygiene regulations (a policy that is law) cover restaurant services independently of anything desired or defined by the restaurant owner.

6.5 Examples

6.5.1 Car Wash Example

See The Washing Policies (Section 8.4) for the **Policy** aspect of the car wash example.

7 Event

7.1 Introduction

Events and the elements that generate or respond to them are important aspects of any event emitting system. SOA systems are in fact often event emitting, hence *event* is defined as a concept in the SOA ontology.

This chapter describes the following classes of the ontology:

- **Event**

In addition, it defines the following properties:

- **generates** and **generatedBy**
- **respondsTo** and **respondedToBy**

7.2 The Event Class

```
<owl:Class rdf:about="#Event">
  <owl:disjointWith>
    <owl:Class rdf:about="#Policy"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#ServiceContract"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#ServiceInterface"/>
  </owl:disjointWith>
</owl:Class>
```

An *event* is something that happens, to which an element may choose to respond. Events can be responded to by any element. Similarly, events may be generated (emitted) by any element. Knowing the events generated or responded to by an element makes it easier and more transparent to interact with that element. Note that some events may occur whether generated or responded to by an element or not. The concept of *event* is captured by the **Event** OWL class, which is illustrated below (in Figure 17).

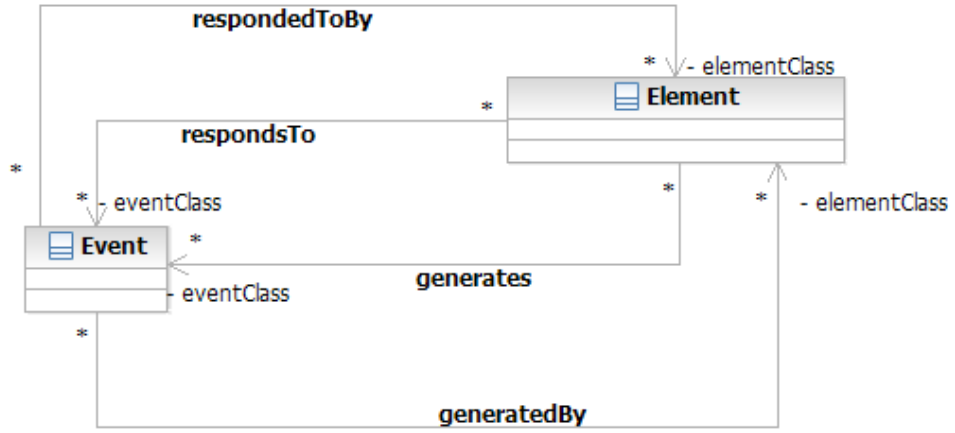

Figure 17: The Event Class

Event as a concept is generic and has relevance to the domain of SOA as well as many other domains. For the purposes of this ontology, event is used in its generic sense.

From a design perspective events may have more granular parts or may be expressed and made operational through specific syntax or semantics. We have chosen to stay at the concept level and not include such design aspects in the ontology.

7.3 The generates and generatedBy Properties

```
<owl:ObjectProperty rdf:about="#generates">
  <rdfs:domain rdf:resource="#Element"/>
  <rdfs:range rdf:resource="#Event"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#generatedBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#generates"/>
  </owl:inverseOf>
</owl:ObjectProperty>
```

Events can, but need not necessarily, be generated by elements. The **generates** property, and its inverse **generatedBy**, capture the abstract notion that an element generates an event.

Note that the same event may be generated by many different elements. Similarly, the same element may generate many different events.

7.4 The respondsTo and respondedToBy Properties

```
<owl:ObjectProperty rdf:about="#respondsTo">
```

```
  <rdfs:domain rdf:resource="#Element"/>
  <rdfs:range rdf:resource="#Event"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#respondedToBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#respondsTo"/>
  </owl:inverseOf>
</owl:ObjectProperty>
```

Events can, but need not necessarily, be responded to by elements. The **respondsTo** property, and its inverse **respondedToBy**, capture the abstract notion that an element responds to an event.

Note that the same event may be responded to by many different elements. Similarly, the same element may respond to many different events.

8 Complete Car Wash Example

This chapter contains the complete car wash example that has been used in parts throughout the definitional chapters of the ontology.

8.1 The Organizational Aspect

Joe the owner chooses to organize his business into two organizational units: **Administration** and **CarWash**:

- **CarWashBusiness** is an instance of both **HumanActor** and **System**.
- **Administration** is an instance of **HumanActor** (organizational unit).
- **CarWash** is an instance of **HumanActor** (organizational unit).
- **CarWashBusiness** uses (has organizational units) **Administration** and **CarWash**.
- **AdministrativeSystem** is an instance of **System**.
- **Administration** represents **AdministrativeSystem**.
- **CarWashSystem** is an instance of **System**.
- **CarWash** represents **CarWashSystem**.

And using well-defined roles within each organization:

- **Owner** (role) is an instance of **Element** and is used by **AdministrativeSystem**.
- **Joe** is an instance of **HumanActor** and is represented by (has role) **Owner**.
- **Secretary** (role) is an instance of **Element** and is used by **AdministrativeSystem**.
- **Mary** is an instance of **HumanActor** and is represented by (has role) **Secretary**.
- **PreWashGuy** (role) is an instance of **Element** and is used by **CarWashSystem**.
- **John** is an instance of **HumanActor** and is represented by (has role) **PreWashGuy**.
- **WashManager** (role) is an instance of **Element** and is used by **CarWashSystem**.
- **WashOperator** (role) is an instance of **Element** and is used by **CarWashSystem**.
- **Jack** is an instance of **HumanActor** and is represented by (has roles) both **WashManager** and **WashOperator**.

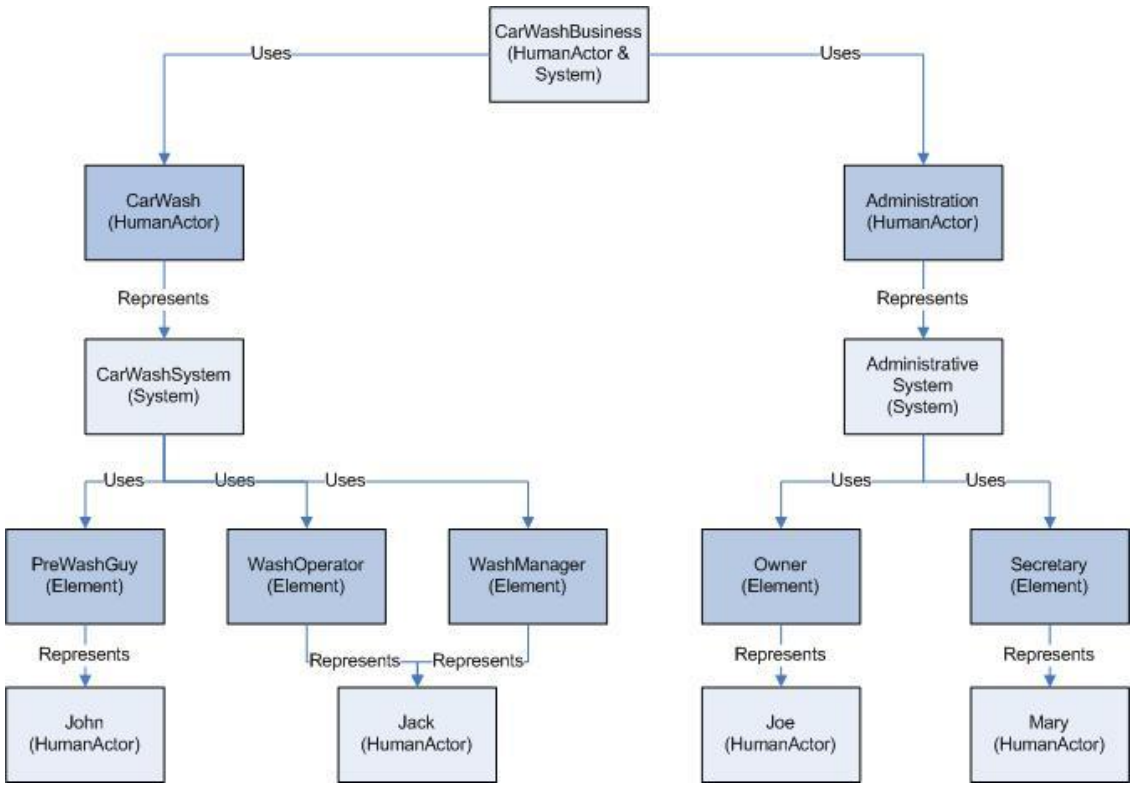

Figure 18: Car Wash Example – The Organizational Aspect

8.2 The Washing Services

Joe offers two different services to his customers: a basic wash and a gold wash:

- **GoldWash** is an instance of **Service**.
- **BasicWash** is an instance of **Service**.
- **CarWash** performs both **BasicWash** and **GoldWash**.
- **WashManager** represents both **BasicWash** and **GoldWash** (i.e., it is the interaction point where customers can order services as well as pay for them).

In return for payment, Joe's **BasicWash** service cleans the car of customer **Judy**:

- **Judy** is an instance of **HumanActor** (the customer).
- **BasicWashContract** is an instance of **ServiceContract**.
- **BasicWash** has contract **BasicWashContract**.
- **CleanCar** is an instance of **Effect**.
- **BasicWashContract** specifies **CleanCar** as its effect.

- **BasicWashContract** involves parties **CarWashBusiness** and **Judy** and specifies that **Judy** (as the legal consumer) pays **CarWashBusiness** (as the legal provider) $10 for the one consumption of **BasicWash** with the effect of (one) **CleanCar**. Note that **BasicWash** is actually performed by **CarWash** and not by the legal provider **CarWashBusiness** – in this particular example **CarWash** happens to be a member of **CarWashBusiness** but such need not always be the case, **CarWash** could have been some third-party provider.
- **Judy** uses **WashManager** (in order to invoke the **BasicWash** service).

Note that in this example Judy does not interact with the (abstract) **BasicWash** service directly, rather she interacts with the **WashManager** that represents the service. This is due to Joe deciding that in his car wash customers are not to interact with the washing machinery directly.

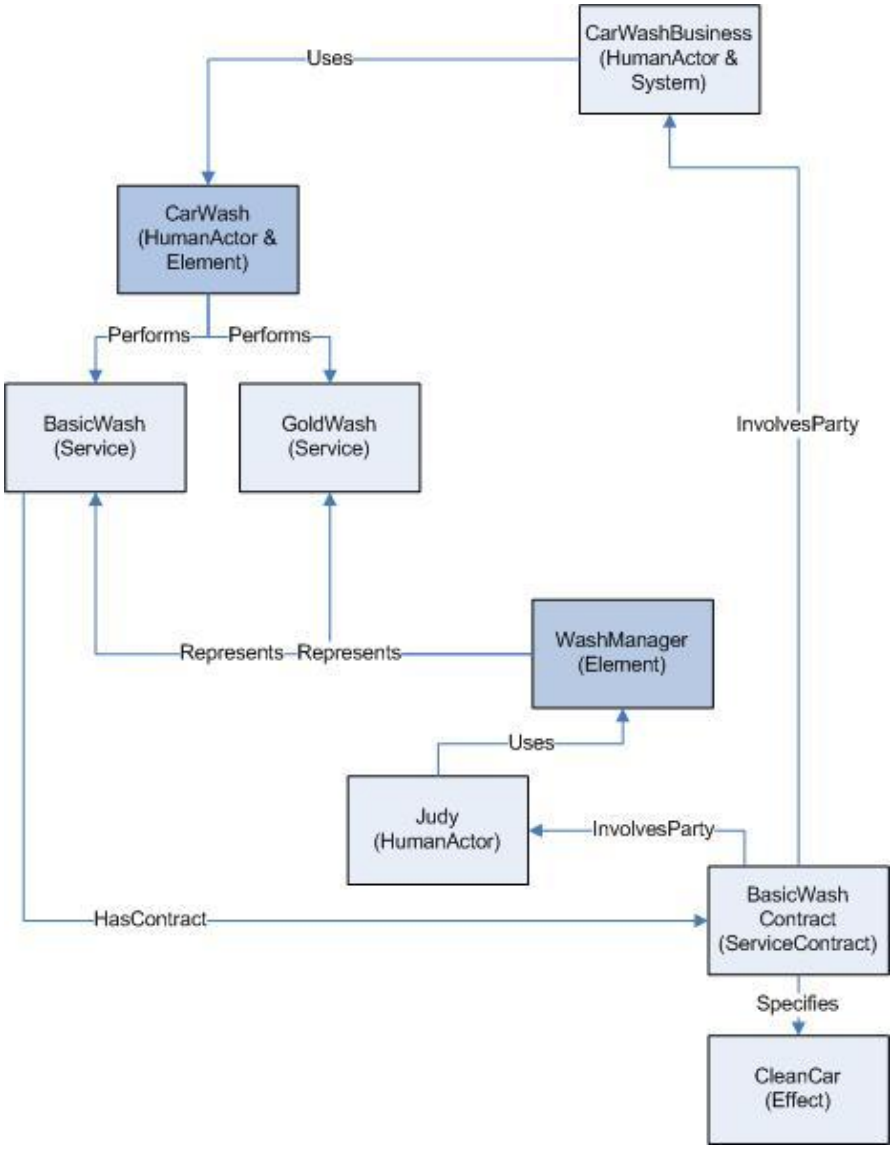

Figure 19: Car Wash Example – The Washing Services

8.2.1 Interfaces to the Washing Services

The way to interact with the car wash services is simple for the customer; he or she simply gives money to the wash manager and asks to have the car washed using one of the two available wash services. Due to the fact that Joe has decided to interpose the wash manager between the customer and the washing machine, the customer actually never interacts with the wash services themselves. We could have chosen to formally define a proxy service provided by the wash manager but have omitted that level of formality in this real-world example.

The wash manager in turn does interact with the wash services through their interfaces defined as follows:

- **WashingMachineInterface** is an instance of **ServiceInterface**.
- **TypeOfWash** is an instance of **InformationType**.
- **WashingMachineInterface** has input **TypeOfWash**.
- **BasicWash** has interface **WashingMachineInterface**.
- **GoldWash** has interface **WashingMachineInterface**.

Note how both washing services in fact have the same service interface. Even though Joe has chosen to offer basic wash and gold wash as two different services, both are in effect done by the same washing machine (one simply has to choose the type of wash when initializing the washing machine).

8.3 The Washing Processes

An important part of the car wash system is the car washing process itself:

- **AutomatedCarWashProcess** is an instance of both **Process** and **Orchestration**.
- **Wash** is an instance of **Task** and is used by **AutomatedCarWashProcess**.
- **Dry** is an instance of **Task** and is used by **AutomatedCarWashProcess**.
- **AutomatedCarWash** is an instance of **Element** (the automated washing machine) and represents **AutomatedCarWashProcess** (encapsulates the process) as well as directs **AutomatedCarWashProcess**.
- **CarWashProcess** is an instance of **Process** and is used by (part of) **CarWashSystem** (no need to create an explicit opaque building block).
- **AutomatedCarWash** is used by **CarWashProcess** (automated activity in the process).
- **WashWindows** is an instance of **Task** and is done by **John**.
- **PreWash** is an instance of **Element**, represents **WashWindows**, and is used by **CarWashProcess** (logical activity in the process).

- **PrewashGuy** is a member of **CarWashProcess** (role in the process).
- **PushWashButton** is an instance of **Task** and is done by **Jack**.
- **InitiateAutomatedWash** is an instance of **Element**, represents **PushWashButton**, and is used by **CarWashProcess** (logical activity in the process).
- **WashOperator** is a member of **CarWashProcess** (role in the process).

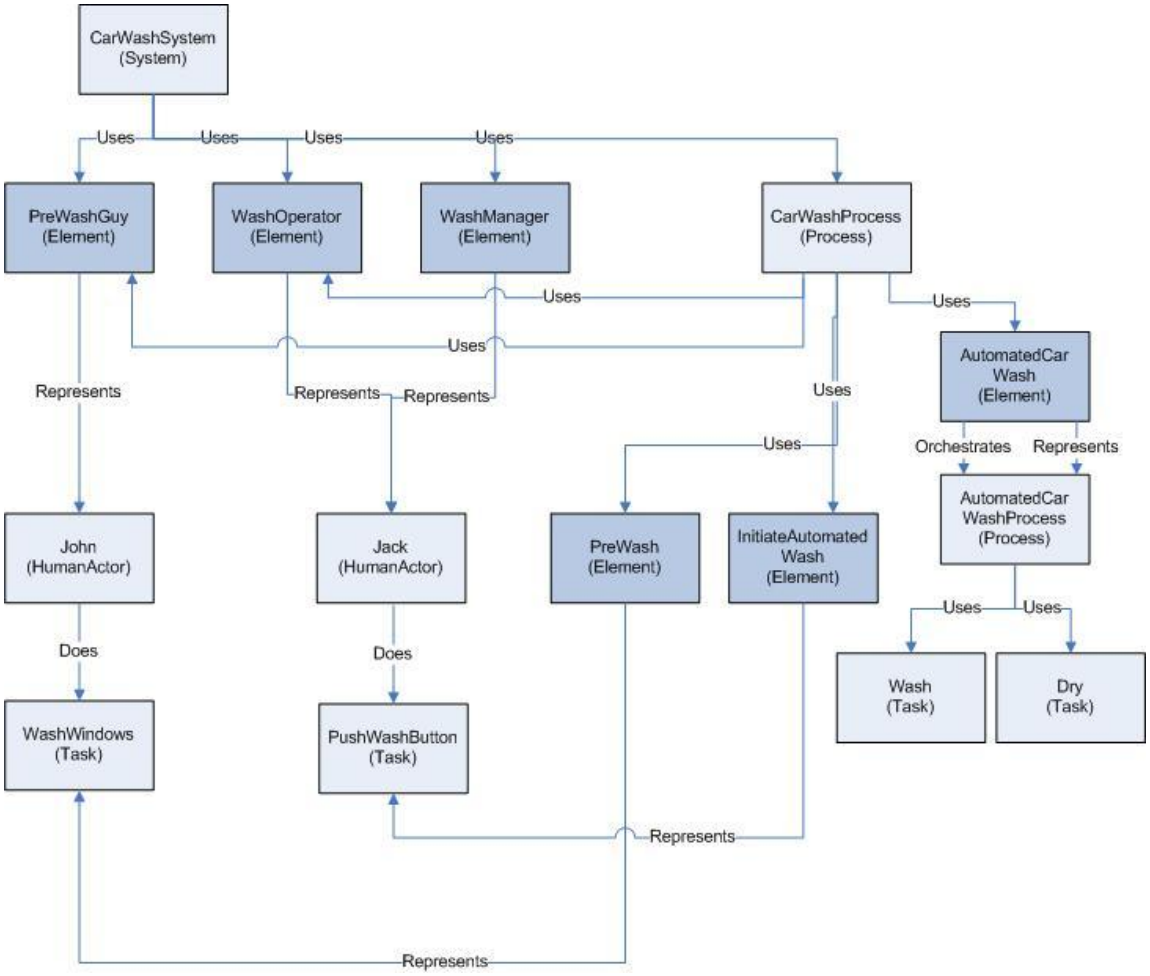

Figure 20: Car Wash Example – The Washing Processes

8.4 The Washing Policies

Joe sets a payment up-front policy for the washing services:

- **PaymentUpFront** is an instance of **Policy**.
- **PaymentUpFront** is set by **Joe**.

- **PaymentUpFront** applies to both **GoldWash** and **BasicWash**.

Note how the **PaymentUpFront** policy enhances the service contract **BasicWashContract**. While **BasicWashContract** only specifies that **Judy** has to pay $10 for one consumption of the **BasicWash** service, the **PaymentUpFront** policy makes it specific that payment has to happen up-front. One of the advantages of separating policy from service contract is that the payment policy can be changed independently of the service contract. For instance, at some later point in time Joe may decide that recurring customers need not pay up-front, and can institute this change in policy without changing anything else related to **CarWashBusiness**.

9 Internet Purchase Example

Jill is purchasing a new TV on the Internet through an online sales site:

- **Jill** is an instance of **Actor** (person).
- **PurchaseTV** is an instance of **Task**.
- **Jill** does **PurchaseTV**.
- **BuyTVOnline** is an instance of **Service**.
- **PurchaseTV** uses **BuyTVOnline**.

OnlineTVSales is the company that is selling TVs:

- **OnlineTVSales** is an instance of **Actor** (organization).
- **BuyTVOnlineContract** is an instance of **ServiceContract** (and describes how to interact with **BuyTVOnline** as well as the legal contract between TV buyer and **OnlineTVSales**).
- **BuyTVOnline** has contract **BuyTVOnlineContract**.
- **OnlineTVSales** is party to **BuyTVOnlineContract**.
- **Jill** is party to **BuyTVOnlineContract**.

The online site is implemented using web site software:

- **OnlineSalesComponent** is an instance of **Element**.
- **OnlineSalesComponent** performs **OnlineTVSales**.
- **SelectWhatToBuyComponent** is an instance of **Element**.
- **SelectWhatToBuyService** is an instance of **Service**.
- **SelectWhatToBuyComponent** performs **SelectWhatToBuyService**.
- **PayComponent** is an instance of **Element**.
- **PayService** is an instance of **Service**.
- **PayComponent** performs **PayService**.
- **OnlineSalesComponent** is also an instance of **ServiceComposition**.
- **OnlineSalesComponent** uses **SelectWhatToBuyService** and **PayService**.

To complete the purchase transaction, Jill needs to pay for the purchase and then the TV will be delivered:

- **PayForTV** is an instance of **Task**.
- **Jill** does **PayForTV**.
- **PayForTV** uses **BuyTVOnline**.
- **DeliverTV** is an instance of **Task**.
- **OnlineTVSales** does **DeliverTV**.
- **OnlineTVSalesProcess** is an instance of **Process**.
- **OnlineTVSalesProcess** uses **Jill**, **OnlineTVSales**, **PurchaseTV**, **PayForTV**, and **DeliverTV**.

A The OWL Definition of the Ontology

The OWL ontology is available online at:

http://www.opengroup.org/soa/ontology/20140404/soa.owl

and is reproduced below.

```
<?xml version="1.0"?>

<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns="http://www.semanticweb.org/ontologies/2010/01/core-soa.owl#"
    xml:base="http://www.semanticweb.org/ontologies/2010/01/core-
soa.owl"
>

  <!-- ontology -->
  <owl:Ontology rdf:about=""/>
  <!-- classes -->

  <owl:Class rdf:about="#Event">
    <owl:disjointWith>
      <owl:Class rdf:about="#Policy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceContract"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceInterface"/>
    </owl:disjointWith>
  </owl:Class>

  <owl:Class rdf:about="#InformationType">
    <owl:disjointWith>
      <owl:Class rdf:about="#Policy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceContract"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Effect"/>
    </owl:disjointWith>
  </owl:Class>
```

```
  <owl:Class rdf:about="#ServiceComposition">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Composition"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceContract"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceInterface"/>
    </owl:disjointWith>
  </owl:Class>

  <owl:Class rdf:about="#Effect">
    <owl:disjointWith>
      <owl:Class rdf:about="#Policy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceInterface"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#InformationType"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#isSpecifiedBy"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:about="#Task">
    <owl:disjointWith>
      <owl:Class rdf:about="#Policy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#System"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#HumanActor"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Service"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceContract"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceInterface"/>
    </owl:disjointWith>
```

```
    <owl:disjointWith>
      <owl:Class rdf:about="#Composition"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Element"/>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#doneBy"/>
        </owl:onProperty>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >0</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:maxCardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#doneBy"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:about="#System">
    <owl:disjointWith>
      <owl:Class rdf:about="#Task"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Service"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Element"/>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:about="#Service">
    <owl:disjointWith>
      <owl:Class rdf:about="#System"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Task"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#HumanActor"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceInterface"/>
    </owl:disjointWith>
```

```
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Element"/>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#hasInterface"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:about="#Policy">
    <owl:disjointWith>
      <owl:Class rdf:about="#InformationType"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceInterface"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Element"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Effect"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Event"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceContract"/>
    </owl:disjointWith>
  </owl:Class>

  <owl:Class rdf:about="#HumanActor">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Element"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Task"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Service"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceContract"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceInterface"/>
    </owl:disjointWith>
  </owl:Class>
```

```
  <owl:Class rdf:about="#Composition">
    <owl:disjointWith>
      <owl:Class rdf:about="#Task"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#System"/>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:maxCardinality>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:about="#compositionPattern"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:about="#compositionPattern"/>
        </owl:onProperty>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:maxCardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#orchestratedBy"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >0</owl:minCardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#orchestratedBy"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:about="#ServiceInterface">
    <owl:disjointWith>
      <owl:Class rdf:about="#Service"/>
```

```
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceContract"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Effect"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Policy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#HumanActor"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Task"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceComposition"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Process"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Event"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:about="#constraints"/>
        </owl:onProperty>
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:maxCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:about="#constraints"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:about="#Element">
    <owl:disjointWith>
      <owl:Class rdf:about="#Policy"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
```

```
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >0</owl:minCardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#orchestrates"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:maxCardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#orchestrates"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:about="#ServiceContract">
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceInterface"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Policy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#HumanActor"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Task"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#ServiceComposition"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Process"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Event"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#InformationType"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:about="#legalAspect"/>
        </owl:onProperty>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
```

```
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:maxCardinality>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:about="#legalAspect"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:about="#interactionAspect"/>
        </owl:onProperty>
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:maxCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:about="#interactionAspect"/>
        </owl:onProperty>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#isContractFor"/>
        </owl:onProperty>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#specifies"/>
        </owl:onProperty>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
```

```
<owl:Class rdf:about="#Process">
  <owl:disjointWith>
    <owl:Class rdf:about="#ServiceContract"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#ServiceInterface"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Composition"/>
  </rdfs:subClassOf>
</owl:Class>

<!-- object properties -->

<owl:ObjectProperty rdf:about="#isPartyTo">
  <rdfs:domain rdf:resource="#HumanActor"/>
  <rdfs:range rdf:resource="#ServiceContract"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#involvesParty">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#isPartyTo"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#orchestratedBy">
  <rdfs:domain rdf:resource="#Composition"/>
  <rdfs:range rdf:resource="#Element"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#orchestrates">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#orchestratedBy"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#isContractFor">
  <rdfs:domain rdf:resource="#ServiceContract"/>
  <rdfs:range rdf:resource="#Service"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasContract">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#isContractFor"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#setsPolicy">
  <rdfs:domain rdf:resource="#HumanActor"/>
  <rdfs:range rdf:resource="#Policy"/>
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:about="#isSetBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#setsPolicy"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#generates">
  <rdfs:domain rdf:resource="#Element"/>
  <rdfs:range rdf:resource="#Event"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#generatedBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#generates"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#represents">
  <rdfs:domain rdf:resource="#Element"/>
  <rdfs:range rdf:resource="#Element"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#representedBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#represents"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasInput">
  <rdfs:domain rdf:resource="#ServiceInterface"/>
  <rdfs:range rdf:resource="#InformationType"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#isInputAt">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#hasInput"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#doneBy">
  <rdfs:domain rdf:resource="#Task"/>
  <rdfs:range rdf:resource="#HumanActor"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#does">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#doneBy"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#specifies">
  <rdfs:domain rdf:resource="#ServiceContract"/>
  <rdfs:range rdf:resource="#Effect"/>
```

```
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#isSpecifiedBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#specifies"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#appliesTo">
  <rdfs:domain rdf:resource="#Policy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#isSubjectTo">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#appliesTo"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#hasInterface">
  <rdfs:domain rdf:resource="#Service"/>
  <rdfs:range rdf:resource="#ServiceInterface"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#isInterfaceOf">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#hasInterface"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#respondsTo">
  <rdfs:domain rdf:resource="#Element"/>
  <rdfs:range rdf:resource="#Event"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#respondedToBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#respondsTo"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#performs">
  <rdfs:domain rdf:resource="#Element"/>
  <rdfs:range rdf:resource="#Service"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#performedBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#performs"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#uses">
  <rdfs:domain rdf:resource="#Element"/>
```

```
    <rdfs:range rdf:resource="#Element"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#usedBy">
    <owl:inverseOf>
      <owl:ObjectProperty rdf:about="#uses"/>
    </owl:inverseOf>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#hasOutput">
    <rdfs:domain rdf:resource="#ServiceInterface"/>
    <rdfs:range rdf:resource="#InformationType"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#isOutputAt">
    <owl:inverseOf>
      <owl:ObjectProperty rdf:about="#hasOutput"/>
    </owl:inverseOf>
  </owl:ObjectProperty>

  <!-- datatype properties -->

  <owl:DatatypeProperty rdf:about="#legalAspect">
    <rdfs:domain rdf:resource="#ServiceContract"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:about="#constraints">
    <rdfs:domain rdf:resource="#ServiceInterface"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:about="#compositionPattern">
    <rdfs:domain rdf:resource="#Composition"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:about="#interactionAspect">
    <rdfs:domain rdf:resource="#ServiceContract"/>
  </owl:DatatypeProperty>

</rdf:RDF>
```

B Relationship to Other SOA Standards

A White Paper – Navigating the SOA Open Standards Landscape Around Architecture – has been written that positions the SOA Ontology with other architectural standards. This joint White Paper from OASIS, OMG, and The Open Group was written to help the SOA community at large navigate the myriad of overlapping technical products produced by these organizations with specific emphasis on the "A" in SOA; i.e., Architecture.

This joint White Paper explains and positions standards for SOA reference models, ontologies, reference architectures, maturity models, modeling languages, and standards work on SOA governance. It outlines where the works are similar, highlights the strengths of each body of work, and touches on how the work can be used together in complementary ways.. It is also meant as a guide to users of these specifications for selecting the technical products most appropriate for their needs, consistent with where they are today and where they plan to head on their SOA journeys.

The following is a summary of the positioning and guidance on the specifications:

- The OASIS Reference Model for SOA (SOA RM) is the most abstract of the specifications positioned. It is used for understanding of core SOA concepts.
- The Open Group SOA Ontology extends, refines, and formalizes some of the core concepts of the SOA RM. It is used for understanding of core SOA concepts and facilitates a model-driven approach to SOA development.
- The OASIS Reference Architecture for SOA Foundation is an abstract, foundation reference architecture addressing the ecosystem viewpoint for building and interacting within the SOA paradigm. It is used for understanding different elements of SOA, the completeness of SOA architectures and implementations, and considerations for cross-ownership boundaries where there is no single authoritative entity for SOA and SOA governance.
- The Open Group SOA Reference Architecture is a layered architecture from the consumer and provider perspective with cross-cutting concerns describing these architectural building blocks and principles that support the realizations of SOA. It is used for understanding the different elements of SOA, deployment of SOA in the enterprise, basis for an industry or organizational reference architecture, implication of architectural decisions, and positioning of vendor products in SOA context.
- The Open Group SOA Governance Framework is a governance domain reference model and method. It is for understanding SOA governance in organizations. The OASIS Reference Architecture for SOA Foundation contains an abstract discussion of governance principles as applied to SOA with particular application to governance across boundaries.

- The Open Group SOA Integration Maturity Model (OSIMM) is a means to assess an organization's maturity within a broad SOA spectrum and define a roadmap for incremental adoption. It is used for understanding the level of SOA maturity in an organization.
- The Object Management Group SoaML Specification supports services modeling UML extensions. It can be seen as an instantiation of a subset of The Open Group SOA Reference Architecture used for representing SOA artifacts in UML.

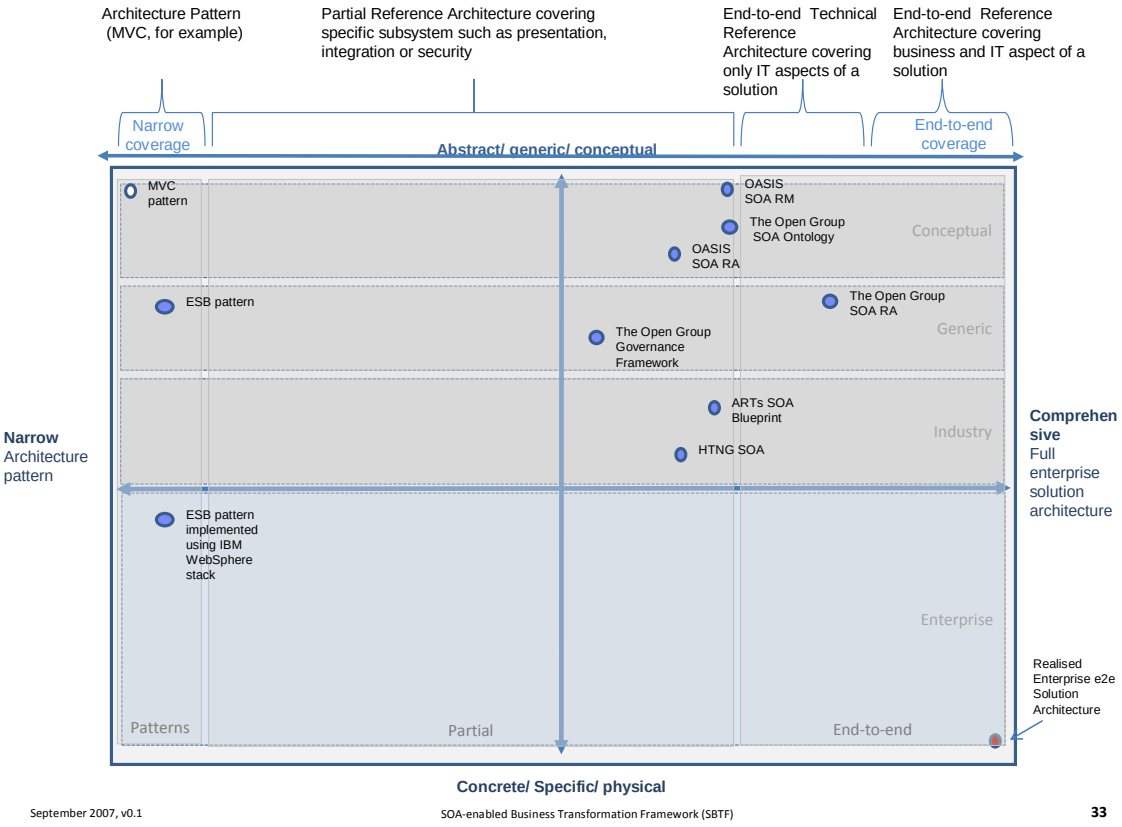

Figure 21: SOA Standards

Fortunately, there is a great deal of agreement on the foundational core concepts across the many independent specifications and standards for SOA. This could be best explained by broad and common experience of users of SOA and its maturity in the marketplace. It also provides assurance that investing in SOA-based business and IT transformation initiatives that incorporate and use these specifications and standards helps to mitigate risks that might compromise a successful SOA solution.

It is anticipated that future work on SOA standards may consider the positioning in this paper to reduce inconsistencies, overlaps, and gaps between related standards and to ensure that they continue to evolve in as consistent and complete a manner as possible.

While the understanding of SOA and SOA Governance concepts provided by these works is similar, the evolving standards are written from different perspectives. Each specification supports a similar range of opportunity, but has provided different depths of detail for the perspectives on which they focus. Therefore, although the definitions and expressions may differ somewhat, there is agreement on the fundamental concepts of SOA and SOA Governance.

C Class Relationship Matrix

This appendix contains a class relationship matrix that illustrates the class-to-class relationships intrinsic in the OWL definitions of the SOA ontology. The matrix is deterministically derived from the OWL ontology definitions. Each row X and each column Y corresponds to an OWL class. A relation appears in cell (X,Y) if and only if class X is part of the domain and class Y is part of the range of the corresponding OWL property. Note that this means that datatype properties (which do not have a range) are not included in the class relationship matrix.

As outlined in the body of the document there are four relationships in the table (plus their inverses and sub-classed derivatives) that are technically allowed according to the OWL definitions, but would not be expected to occur in a practical application of the ontology. Specifically, services are not expected to perform services, services are not expected to use elements (directly), services are not expected to represent elements, and services are not expected to orchestrate compositions – all due to the **Service** class being defined as a logical representation of a repeatable activity; see The performs and performedBy Properties (Section 4.3), The uses and usedBy Properties Applied to Service (Section 4.4.1), The represents and representedBy Properties Applied to Service (Section 4.4.2) and The orchestrates and orchestratedBy Properties (Section 5.3) for details.

	Element	**System**	**Service**	**Human Actor**	**Task**
Element	uses usedBy represents representedBy	uses usedBy represents representedBy	uses usedBy represents representedBy performs	uses usedBy represents representedBy	uses usedBy represents representedBy
System	uses usedBy represents representedBy	uses usedBy represents representedBy	uses usedBy represents representedBy performs	uses usedBy represents representedBy	uses usedBy represents representedBy
Service	uses usedBy represents representedBy performedBy	uses usedBy represents representedBy performedBy	uses usedBy represents representedBy performs performedBy	uses usedBy represents representedBy performedBy	uses usedBy represents representedBy performedBy
Human Actor	uses usedBy represents representedBy	uses usedBy represents representedBy	uses usedBy represents representedBy performs	uses usedBy represents representedBy	uses usedBy represents representedBy does

	Element	System	Service	Human Actor	Task
Task	uses usedBy represents representedBy	uses usedBy represents representedBy	uses usedBy represents representedBy performs	uses usedBy represents representedBy doneBy	uses usedBy represents representedBy
Composition	uses usedBy represents representedBy orchestratedBy	uses usedBy represents representedBy orchestratedBy	uses usedBy represents representedBy performs orchestratedBy	uses usedBy represents representedBy orchestratedBy	uses usedBy represents representedBy orchestratedBy
Process	uses usedBy represents representedBy orchestratedBy	uses usedBy represents representedBy orchestratedBy	uses usedBy represents representedBy performs orchestratedBy	uses usedBy represents representedBy orchestratedBy	uses usedBy represents representedBy orchestratedBy
Service Composition	uses usedBy represents representedBy orchestratedBy	uses usedBy represents representedBy orchestratedBy	uses usedBy represents representedBy performs orchestratedBy	uses usedBy represents representedBy orchestratedBy	uses usedBy represents representedBy orchestratedBy
Service Contract			isContractFot	involvesParty	
Effect					
Service Interface			isInterfaceOf		
Information Type					
Event	generatedBy respondedToBy	generatedBy respondedToBy	generatedBy respondedToBy	generatedBy respondedToBy	generatedBy respondedToBy
Policy	appliesTo	appliesTo	appliesTo	isSetBy appliesTo	appliesTo
Thing					

	Composition	Process	Service Composition	Service Contract	Effect
Element	uses usedBy represents representedBy orchestrates	uses usedBy represents representedBy orchestrates	uses usedBy represents representedBy orchestrates		

	Composition	Process	Service Composition	Service Contract	Effect
System	uses usedBy represents representedBy orchestrates	uses usedBy represents representedBy orchestrates	uses usedBy represents representedBy orchestrates		
Service	uses usedBy represents representedBy performedBy orchestrates	uses usedBy represents representedBy performedBy orchestrates	uses usedBy represents representedBy performedBy orchestrates	hasContract	
Human Actor	uses usedBy represents representedBy orchestrates	uses usedBy represents representedBy orchestrates	uses usedBy represents representedBy orchestrates	isPartyTo	
Task	uses usedBy represents representedBy orchestrates	uses usedBy represents representedBy orchestrates	uses usedBy represents representedBy orchestrates		
Composition	uses usedBy represents representedBy orchestrates orchestratedBy	uses usedBy represents representedBy orchestrates orchestratedBy	uses usedBy represents representedBy orchestrates orchestratedBy		
Process	uses usedBy represents representedBy orchestrates orchestratedBy	uses usedBy represents representedBy orchestrates orchestratedBy	uses usedBy represents representedBy orchestrates orchestratedBy		
Service Composition	uses usedBy represents representedBy orchestrates orchestratedBy	uses usedBy represents representedBy orchestrates orchestratedBy	uses usedBy represents representedBy orchestrates orchestratedBy		
Service Contract					specifies
Effect				isSpecifiedBy	
Service Interface					

	Composition	Process	Service Composition	Service Contract	Effect
Information Type					
Event	generatedBy respondedToBy	generatedBy respondedToBy	generatedBy respondedToBy		
Policy	appliesTo	appliesTo	appliesTo	appliesTo	appliesTo
Thing					

	Service Interface	Information Type	Event	Policy	Thing
Element			generates respondsTo	isSubjectTo	
System			generates respondsTo	isSubjectTo	
Service	hasInterface		generates respondsTo	isSubjectTo	
Human Actor			generates respondsTo	setsPolicy isSubjectTo	
Task			generates respondsTo	isSubjectTo	
Composition			generates respondsTo	isSubjectTo	
Process			generates respondsTo	isSubjectTo	
Service Composition			generates respondsTo	isSubjectTo	
Service Contract				isSubjectTo	
Effect				isSubjectTo	
Service Interface		hasInput hasOutput		isSubjectTo	
Information Type	isInputAt isOutputAt			isSubjectTo	
Event				isSubjectTo	
Policy	appliesTo	appliesTo	appliesTo	appliesTo isSubjectTo	appliesTo
Thing				isSubjectTo	

Index